NASA-TM-108139

86-FM-25

JSC-22308

CLIPS User's Guide

P-148

Mission Support Directorate
Mission Planning and Analysis Division

October 1986

NASA
National Aeronautics and Space Administration

Lyndon B. Johnson Space Center
Houston, Texas

86-FM-25 JSC-22038

CLIPS USER'S GUIDE

By: Joseph C. Giarratano, Ph.D
Artificial Intelligence Section,
Technology Development and Applications Branch

Approved: [illegible]
Robert H. Brown
Technology Development and Applications Branch

Approved: [illegible]
Edgar C. Lineberry
Mission Planning and Analysis Division

Mission Planning and Analysis Division

National Aeronautics and Space Administration
Lyndon B. Johnson Space Center
Houston, Texas

October 1986

CLIPS User's Guide

Joseph C. Giarratano, Ph.D

FIGURES

Preface

This book is a tutorial on the CLIPS expert system language. Its companion volume is the CLIPS Reference Manual, written by Chris Culbert.

The CLIPS User's Guide can be used in the classroom or for self-teaching. The only prerequisite is that you have a basic knowledge of programming in a high-level language such as Pascal, Ada, FORTRAN, C, or BASIC.

The CLIPS User's Guide is designed for people who want to learn about expert system programming in a hands-on manner. Other than a knowledge of basic programming, no special background is assumed about the reader. Thus, the examples are of a general nature. Also, since learning a new language can be a frustrating experience, the writing is in a light style (I hope).

For maximum benefit, you should type in the example programs in the text as you read through the book. By typing in the examples, you will see how the programs should work and what error messages occur if you make a mistake. The output for the examples is shown or described after each one. You should also plan to work the problems at the end of the chapters. Most of the problems are simply exercises to give you experience in the CLIPS syntax described in each chapter.

Like any other programming language, you will only learn programming in CLIPS by writing programs in it. To really learn expert system programming, you should pick a problem of interest to you and write it in CLIPS. By the time you finish working through this book and doing the problems at the end of the chapters, you should be able to write simple expert systems applications. As your experience with CLIPS grows, you will be able to write more extensive programs.

I greatly appreciate the advice and criticism of several people at Johnson Space Center in the development of this book. Thanks to Gary Riley, Chris Culbert, Brian Donnell, Kirt Fields, and Steven Lewis. Special thanks to Bob Savely for supporting the development of CLIPS and his encouragement of this book.

CLIPS User's Guide

by

Joseph C. Giarratano, Ph.d

Artificial Intelligence Section
NASA--Johnson Space Center

Chapter 1 Just The Facts

In this chapter, you'll learn the basic concepts of an expert system. You'll also see how to insert and remove facts in CLIPS.

Introduction

CLIPS is a type of computer language designed for writing applications in Expert Systems. The basic elements of CLIPS are:

1. **fact-list:** global memory for data
2. **production memory:** contains all the **rules** or **productions.**
3. **inference engine:** controls overall production execution

A program written in CLIPS consists of rules and facts. The inference engine decides which rules should be executed. We'll discuss the parts of CLIPS in more detail in later chapters.

Right now, let's take a look now at the facts of CLIPS. Facts are very important because execution in CLIPS cannot proceed without facts. This is one example in which CLIPS is different from procedural languages such as Pascal, Ada, BASIC, FORTRAN and C. In procedural languages, execution can proceed without facts. That is, the statements are sufficient in those languages to cause exection. However, in CLIPS, facts are required to cause the execution of rules and so play a very important role.

The Beginning And The End

To begin CLIPS, just enter the appropriate run command for your system. You should see the CLIPS prompt appear as follows.

```
CLIPS>
```

At this point, you can start entering commands to CLIPS.

The normal mode of leaving CLIPS is with the **exit** command. Just type

```
(exit)
```

in response to the CLIPS prompt and then press the Return key.

Making A List

As with other programming languages, CLIPS recognizes certain keywords. For example, if you want to put data in the fact-list, you can use the **assert** command.

As an example of assert, enter the following right after the CLIPS prompt, as shown.

```
CLIPS>(assert (duck))
```

After every line, always press the Return key to send the line to CLIPS. The above command will put the fact "duck" in fact-memory.

Notice that the assert command and the (duck) fact are surrounded by parentheses. CLIPS, like many other expert system languages, has a LISP-like syntax which requires that all statements be surrounded by parentheses. Even though CLIPS is not written in LISP, the style of LISP has influenced the development of CLIPS. This is also why the exit command requires parentheses.

And Checking It Twice

Suppose you want to see what's in the fact-list. Just use the **facts** command. Enter (facts) in response to the CLIPS prompt and CLIPS will respond with a list of the facts in fact-memory. Be sure to put parentheses around the command or CLIPS will not accept it. The results of the (facts) command in this example should be

```
f-1     (duck)
```

The term "f-1" is the **fact identifier** put in by CLIPS. Every fact that is inserted into the fact-list is assigned a unique fact identifier starting with the letter f and followed by an integer called the **fact-index.**

What happens if you try to put a second duck into the fact-list? Let's try it and see. Enter a new (duck) and then issue a (facts) command. You'll see just the original "f-1   (duck)". This shows that CLIPS will not accept a duplicate entry of a fact.

Of course, you can put other facts in that are different. For example, enter a (quack) fact and then issue a (facts) command. You'll see

```
f-1     (duck)
f-2     (quack)
```

As you can see, the (quack) fact is in the fact-list. An important point to realize is that identifiers in the fact-list may not be strictly sequential. You'll see in a later section that just as you can add facts to the fact-list, there is also a way to remove facts. As facts are removed from the fact-list, the deleted fact identifiers will be missing in the fact-list. So the fact-identifiers may not be stricly sequential as your program executes.

Sensitive Atoms

A fact such as (duck) or (quack) is said to consist of a single **atom.** In LISP-like languages, an atom is the smallest unit of meaning and cannot be split up into smaller pieces. In languages such as Pascal, Ada, C, BASIC and others, a fact such as (duck) could be split up into individual characters or groups of characters. However, (duck) cannot be split up any further in CLIPS because it is an atom.

A **symbolic atom** is a type of atom starting with an alphabetic character and optionally followed by letters, numbers, and dashes. For example, all the words in this sentence are atoms. Another type of atom is the **literal atom.** A literal atom must begin and end with double quotes. The quotes are part of the atom. There can be zero or more characters between the double quotes. The third type of atom is the **numeric atom,** which represents a floating point number. All numbers in CLIPS are treated as floating point. A **string** can be either a symbolic or literal atom.

A **fact** consists of one or more atoms, enclosed in matching left and right parentheses. Multiple atoms must be separated by one or more spaces. For example, enter the following command.

```
(assert (The duck said "Quack."))
```

and then issue a (facts) command. You'll see the group of atoms (The duck said "Quack.") asserted as a single fact since only one identifier is assigned.

What happens if you use multiple spaces with atoms? Assert

```
(   duck   quack   quack   )
```

and then check the fact-list. You'll see

```
(duck quack quack)
```

because CLIPS has eliminated unnecessary spaces around atoms.

Also, notice that CLIPS preserved the upper-case and lower-case letters in the atoms of the fact. That is, the T of The and the Q of Quack are left as upper-case. CLIPS is said to be **case-sensitive** because it distinguishes between upper- and lower-case letters. As an example, assert the facts (duck) and (Duck) and then issue a (facts) command. You'll see that CLIPS allows you to assert (duck) and (Duck) because CLIPS is case-sensitive.

Getting Spaced Out

Since spaces are used to separate multiple atoms, it follows that spaces cannot be simply included in facts. For example, assert the facts (duck), (duck), (duck) and (duck   ). You will only see (duck) since CLIPS ignores spaces as part of the atom and considers all these facts as equivalent to (duck).

If you want to include spaces in a fact, you must use double quotes. For example, assert

```
("duck")
("duck ")
(" duck")
(" duck ")
```

and check the fact-list with a (facts) command. You'll see that all of these facts have been put in the fact-list.

What if you want to include the double quotes themselves in a fact? At first, you might think you could quote the quotes. Let's try it and see what happens. Assert the fact

```
(""duck"")
```

and check the fact-list. You'll see a fact

```
("" duck "")
```

While this fact resembles the assertion, it is actually quite different. Although the double quotes did appear in the fact, they are not around the duck fact anymore. The clue to this is the space before and after the duck fact that was put in by CLIPS. Notice that the asserted fact did not have spaces around duck. The spaces were put around duck because CLIPS thinks you are asserting the three atoms

```
""
duck
""
```

and so CLIPS puts spaces around duck to separate the three atoms.

The correct way to put double quotes in a fact is with the **backslash** operator, \. For example, assert the fact

```
("\"duck\"")
```

and check the fact-list. Now you'll see the correct fact

```
(""duck"")
```

with no spaces added by CLIPS.

Retract That Fact

Now that you know how to put facts into the fact-list, it's time to learn how to remove facts. Removing facts from the fact-list is called retraction and is done with the **retract** command. To retract a fact, you must specify the fact index of the fact as the argument of retract. For example, if your fact-list is

```
f-1    (duck)
f-3    (quack)
f-4    (The duck said "Quack.")
```

To remove the (quack), enter the command

```
(retract 3)
```

and then check the fact-list with a (facts) command. You'll see that (quack) has been retracted. To retract (duck), enter

```
(retract 1)
```

and to retract (The duck said "Quack."), enter

```
(retract 4)
```

Notice that to retract a fact you must specify the fact index and not the position of the fact in the fact-list. A potential error you can make is trying to retract the first fact in the fact-list with a (retract 1) command. This will only work correctly if the first fact has a fact index of 1.

What happens if you try to retract a fact that isn't there? Enter a (retract 4) command again and CLIPS will respond with an error message

```
fact 4 does not exist
```

The moral is that you can't take back what you haven't given.

Clearing Up The Facts

If you want to retract specific facts, the (retract) command is very convenient. However, there will be times that you want to retract all the facts in the fact-list. Rather than your having to type in the fact index of each command, you can use the **clear** command. Just enter

(clear)

and all the facts in the fact-list will be removed.
The (clear) restores the original state of CLIPS just as if you first started up CLIPS. To see this, assert (duck) and then check the fact-list. Notice that (duck) has a fact-identifier of f-1 because the (clear) command reset the fact identifiers. The (clear) command actually does more than just remove facts. For example, it also removes all the rules, as you'll see later. As you learn more about CLIPS, you'll understand more of what the (clear) command does.

Problem

Are the following valid or invalid facts? Check to see if you're right by trying to assert them. If valid, identify the type of atoms each fact contains. Note that on some of the examples, CLIPS will hang up until you provide the necessary characters such as closing quotation marks or parentheses. If you can't figure out the correct characters, you'll have to break off execution of CLIPS with a Control C command and then restart CLIPS.

```
(t)
(this)
(this is a test)
(but what is th!!!)
(**!!)
(and so he ate 19 hamburgers)
(1 2 3 4.5 -.0012)
(Al)
(start-fact)
(the---end)
(^&)
(duck")
("")
(John loves Mary)
(" ")
(1000 Main Street)
(1000 Main St.)
("duck)
()
(100 -25.5)
(1e10 10000)
(())
(
)(
(duck (quack))
```

Chapter 2 Following The Rules

In the previous chapter you learned about facts. Now you'll see how the rules of an expert system utilize facts in making a program execute.

Making Good Rules

In order to accomplish useful work, an expert system must have rules as well as facts. Now that you've seen how facts are asserted and retracted, it's time to see how rules work.

A rule is analogous to an IF THEN statement in a language such as Ada, Pascal, FORTRAN, BASIC, or C. An IF THEN rule can be expressed in a mixture of natural language and computer language as follows.

IF certain conditions are true
THEN execute the following actions

Another term for the above statement is **pseudocode**, which literally means false code. More specific pseudocode statements are a useful aid to the programmer in designing and documenting rules.

In computer languages such as Ada, Pascal, FORTRAN, BASIC, and C, there are IF THEN keywords which are recognized by the language. However, since CLIPS has a LISP-like syntax, there are no IF THEN keywords in CLIPS. However, the translation of rules from natural language to CLIPS is not very difficult if you keep the IF THEN analogy in mind. As your experience with CLIPS grows, you'll soon find that writing rules in CLIPS becomes easy.

You can either type rules directly into CLIPS or load rules in from a file of rules created by a word processor. You can use any type of word processor so long as it does not introduce formatting characters into your source code. Formatting characters will not be accepted by CLIPS, so be sure your word processor is in non-document mode when you type in the source code of your program.

Initially, let's see how to type rules directly into CLIPS. The pseudocode for this rule is

IF there is a duck THEN make a quack

The following is the rule expressed in CLIPS syntax. The only additional feature is the term in double quotes, which is an optional description of the rule. To enter the rule, just type it in after the CLIPS prompt and then press the Return key.

(defrule duck-quack "The First Rule" (duck) => (assert (quack)))

If you type in the rule correctly as shown, you should see the CLIPS prompt reappear. Otherwise, you'll see an error message. If you get an error message, most likely you misspelled a keyword or left out a parenthesis. Remember, the number of left and right parentheses must always match in a statement.

Shown following is the same rule with numbers below the rule keyed to an explanation of the parts of the rule.

```
(defrule duck-quack "The First Rule" (duck) => (assert (quack)))
 |< 2 >|<   3    >|<       4        >|< 5 >|<6>|<     7     >|
|<                         1                                  >|
```

A rule consists of the following parts:

1. The entire rule must be surrounded by parentheses.
2. The rule must start with a **defrule** keyword.
3. Following defrule must be the name of the rule. The name can be any valid symbolic atom. If you type in a rule name that is the same as an existing rule, the new rule replaces the old rule. In this rule, the rule name is duck-quack.
4. Next is an optional comment within double quotes. The comment is normally used to describe the purpose of the rule or any other information the programmer desires.
5. After the optional comment are one or more **conditional elements** or **patterns**. Each conditional element consists of one or more **fields**. In the duck-quack rule, the conditional element is (duck) and the one field is duck. An example of a conditional element with two fields is (duck quack).

CLIPS attempts to match conditional elements of rules against facts in the fact-list. If all the conditional elements of a rule match facts, the rule is **activated** and put on the **agenda.** The agenda is a collection of activated rules. There may be zero or more rules in the agenda.

6. The symbol => that follows the conditional elements in a rule is called an **arrow.** The arrow is a symbol representing the

THEN part of an IF THEN rule.

7. The last part of a rule is the list of actions that will be executed when the rule **fires**. The term fires means that CLIPS has selected a certain rule for execution from the agenda. While there may be multiple rules in the agenda, CLIPS fires the rule with the highest priority. In our example, the one action is to assert the fact (quack).

The part of the rule on the left of the arrow is called the left-hand side (**LHS**) and the part on the right of the arrow is called the right-hand side (**RHS**).

No Quack

A rule fires when it has the highest priority of all rules in the agenda. It follows that if there is only one rule in the agenda, that rule will fire. Since we have the duck-quack rule and its conditional element is satisfied by the fact (duck), then the duck-quack rule should fire.

To make a program run, just enter the **run** command. Type (run) and press the Return key in response to the CLIPS prompt.

Nothing happens.

In case you're wondering, no, CLIPS is not all messed up. The reason CLIPS does not fire the rule has to do with the way CLIPS is designed. The design of CLIPS is such that rules only see facts that have been entered after the rules. Remember that you asserted (duck) first when you learned about (clear) in the previous section. Then you typed in the duck-quack rule. Since (duck) was entered before the duck-quack rule, the rule doesn't see the fact and is not put on the agenda for firing. Because duck-quack is not on the agenda, it can't be fired.

You can check what's in the agenda with the **agenda** command. Type (agenda) and press Return in response to the CLIPS prompt. In this case, nothing will be printed since there's nothing in the agenda and CLIPS will just respond with a prompt.

A Quack At Last

OK, it's time now to get serious and fire this crazy rule. In order to fire it, we've got to assert a (duck) after the rule has been entered. At first thought, this seems trivial since you just need to assert (duck). However, if you try this and (run) again, the rule still won't fire. This may be somewhat frustrating. However, before you do something drastic to ease your frustration, like getting married or kicking your pet duck, just think a minute.

Since there is already a (duck) in the fact-list, typing in a new (duck) doesn't do anything. CLIPS will not accept the new (duck) because there is already a (duck). So, what's the solution?

The solution is to retract the first (duck) and then assert the new duck. Once you've done this, issue an (agenda) command again. You'll see

```
0    duck-quack   f-1.
```

The 0 indicates this is the priority of the rule in the agenda. You'll learn about priority in the next chapter. Next comes the name of the rule followed by the fact identifiers that match the rule. In this case, there is only one fact identifier, f-1.

Now type the (run) command and you'll see the rule fire at last. If you check the fact-list, you'll see that there is a quack at last.

Nevermore, Nevermore

An interesting question may occur to you at this time. What if you (run) again ? There is a rule and there is a fact which satisfies the rule and so the rule should fire again. Try a (run) command and you'll see that nothing happens. If you check the agenda, you'll see that the rule didn't fire because there was nothing on the agenda.

The rule didn't fire again because of the way that CLIPS is designed. CLIPS was programmed with a characteristic of a nerve cell called a **neuron.** After a neuron transmits a nerve impulse (fires), no amount of stimulation will make it fire for some time. This phenomenon is called **refraction** and is very important in expert systems.

Without refraction, expert systems would be always caught in trivial loops. That is, as soon as a rule fired, it would keep on firing on that same fact over and over again. In the real world, the stimulus that caused the firing would eventually disappear. For example, the duck would eventually swim away or fly south for the winter. However, in the computer world, once a fact is entered in the fact-list, it stays there until explicitly removed or the power is turned off.

So how do you get the rule to fire again ? Just retract the fact and assert it again. It's as if the old duck flew away and a new one came to trigger the rule again. Basically, CLIPS remembers the fact identifiers that triggered a rule into firing and will not activate that rule again with the same fact identifiers. Of course, a rule may be put on the agenda with the same fact identifiers many times. However, being put on the agenda is not the same as being fired. Only rules that have been fired experience refraction, not rules on the agenda.

Show Me The Rules

Very often you'd like to see the rules while you're in CLIPS. There's a command that shows the rules called **pprule,** which stands for pretty print rule. To see a rule, specify the rule name as argument. For example, enter the following in response to a CLIPS prompt and press Return.

```
(pprule duck-quack)
```

You'll see

```
(defrule duck-quack "The First Rule"
   (duck)
   =>
   (assert (quack)))
```

CLIPS puts different parts of the rule on different lines for the sake of readability for you. You can enter rules on one line, if there's enough room, or multiple lines. From now on, we'll show rules on multiple lines for the sake of readability.

Using a word processor is very convenient for typing programs because of all the editing commands that are available. The terms before the arrow are still considered the LHS and those after the arrow are the RHS of the rule.

What if you want to print a rule but can't remember the name of the rule? No problem. Just use the **rules** command in response to a CLIPS prompt and CLIPS will print out the names of all the rules. For example, enter

```
(rules)
```

and the duck-quack name will be printed since it's the only rule name.

Getting Loaded

You can load a file of rules made on a word processor into CLIPS using the **load** command. Let's assume you have made the duck rule on a word processor and stored it in a file called duck.clp on drive b. Of course, the actual device that you use for file storage will be system dependent and so this example should be taken only as a guide. To load the rule into CLIPS from the file duck.clp on drive b, enter the following in response to the CLIPS prompt.

```
(load "b:duck.clp")
```

If you have typed the file using a word processor and loaded it in, you may see an error message from CLIPS like

```
Illegal construct defrule
Unidentified input character  (ASCII 229) found
```

The problem is that you used document mode in typing the rule in the word processor and so formatting characters like soft returns (ASCII character code 229) were put in. CLIPS is designed to accept only standard characters. Be sure to type CLIPS programs in non-document mode to avoid error messages like the above.

You can use multiple files to save rules. That is, you don't have to store all your rules in one big file. It does take time to read rules into CLIPS and if you have hundreds of rules, it could take a long time.

CLIPS also provides the opposite of the (load) command. With the **save** command, you can save rules in CLIPS to a disk file. For example, to save the duck-quack rule to a file called duck.clp on drive b:, just enter the following in response to a CLIPS prompt.

```
(save "b:duck.clp")
```

The (save) command will save all the rules in CLIPS to the specified file. It is not possible to save specified rules to a file.

Comments, Anyone

It's a good idea to include comments in your CLIPS program. Sometimes, rules can be difficult to understand and comments can be used to explain to the reader what the rule is doing. Comments are also used for good documentation of a program and will be very helpful in lengthy programs.

A comment in CLIPS is any text that begins with semicolon and ends with a Return. The following is an example of comments in the duck program.

```
;************************************
;*                                  *
;* Programmer: J. Giarratano        *
;*                                  *
;* Title      : The Duck Program    *
;*                                  *
;* Date       : 7/7/86              *
;*                                  *
;************************************
;
;The purpose of this rule is to
;make a quack if there is a duck
(defrule duck-quack "The First Rule" ;IF
   (duck)                            ;there is a duck
   =>                                ;THEN
   (assert (quack)))                 ;quack
```

If you type this in a word processor and then load it into CLIPS, you'll notice that all the comments are eliminated in the CLIPS program. That is, CLIPS will ignore the program comments as it's loading the program.

Write To Me

Besides asserting facts in the RHS of rules, you can also print out information using the **printout** keyword. For example, enter a (clear) command and then the following.

```
(defrule duck-quack
   (duck)
=>
   (printout "quack"))
```

The argument of (printout) is enclosed in double quotes, as shown.
Assert a new (duck) and then enter a (run) command. You'll see the output

```
quack1 rules fired
```

where the quack is from the (printout) statement and the informational message "1 rules fired" is from CLIPS.
CLIPS has a carriage return keyword called **crlf** which is very useful in improving the appearance of output by formatting it on different lines. For a change, the crlf is not included in parentheses. The following example of crlf shows how it's used to separate the program output from the informational message of CLIPS.

```
(defrule duck-quack (duck) => (printout "quack" crlf))
```

When you enter a (run) command, you'll see

```
quack
1 rules fired
```

Notice how the quack is separated now from the CLIPS message.

Problems

1. Write a program that will print your name and address when you assert your first name. For example, if you enter (assert (Bob)), the program could print out

```
Bob Jones
1000 Main St.
Houston, TX 77058
```

2. Write a program that will print out the following if an assertion of (holidays) is made.

```
       *
      * *
     *   *
    *     *
      | |
 Happy Holidays
```

Chapter 3 Adding Details

In the first two chapters, you learned the fundamentals of CLIPS. Now you will see how to build on that foundation to create more powerful programs.

Stop And Go

Until now you've only seen the simplest type of program consisting of just one rule. However, expert systems consisting of only one rule are not too useful. Practical expert systems may consist of hundreds or thousands of rules. Let's take a look now at an application requiring multiple rules.

Suppose you wanted to write an expert system to simulate how a robot should respond to a traffic light. This type of problem requires multiple rules. For example, the rules for the red and green light situations can be written as follows.

```
(defrule red-light
   (red)
=>
   (printout "Stop" crlf))

(defrule green-light
   (green)
=>
   (printout "Go" crlf))
```

To test out this program, you can either enter the rules with a word processor and (load) them into CLIPS, or type in each rule in response to a CLIPS prompt.

After the rules have been entered to CLIPS, assert a (red) fact and run. You'll see "Stop" printed. Now assert a (green) fact and run. You should see "Go" printed.

An alternative way of writing the conditional elements in the rules is as follows. By including the extra pattern "light", it is more clear to someone reading the program what is being matched. The only disadvantage is a slightly longer execution time by CLIPS because more work is required to find matches.

```
(defrule red-light
   (light red)
=>
   (printout "Stop" crlf))

(defrule green-light
   (light green)
=>
   (printout "Go" crlf))
```

The Real Thing

As long as our robot encounters only red and green lights, everything is alright. However, the real world is not quite that simple. In addition to red and green lights, there are also yellow lights. In order to make the program more realistic, we need to add a rule for yellow lights. But is that enough ?

Other possibilities include blinking red and blinking yellow lights. Rules need to be made to cover these situations also. Now have we covered all possibilties ?

Unfortunately, the answer is no. In the real world, things don't always operate perfectly. What if the light is blinking green ? What if none of the lights are on ? What if the lights don't change ? What if the light is red but a police officer waves you on ?

Take A Walk

If you think about it, there are other possibilities besides the simple red, green, and yellow cases. Some traffic lights also have a green arrow for protected left turns. Some have a hand that lights up to indicate whether a person can walk or not. Some have signs that say walk or don't walk. So depending on whether our robot is walking or driving, it may have to pay attention to different signs.

The information about walking or driving must be asserted in addition to information about the status of the light. Rules can be made to cover these conditions but they must have more than one conditional element. For example, suppose we want a rule to fire if the robot is walking and if the walk-sign says walk. A rule could be written as follows.

```
(defrule take-a-walk
   (status walking)
   (walk-sign walk)
=>
   (printout "Go" crlf))
```

The above rule has two conditional elements. Both elements must be satisfied by facts in the fact-list for the rule to fire. To see how this works, enter the rule and then assert the facts (status walking) and (walk-sign walk). When you (run), the program will print out "Go" since both conditional elements are satisfied and the rule is fired.

You can have any number of conditional elements or actions in a rule. The important point to realize is that the rule is placed on the agenda only if all the conditional elements are satisfied by facts. This type of restriction is called a **logical AND** in reference to the AND relation of Boolean logic. An AND relation is said to be true only if all its conditions are true.

Because the conditional elements are of the logical AND type, the rule will not fire if only one of the conditional elements are satisfied. All the facts must be present before the LHS of a rule is satisfied and the rule is placed on the agenda.

Gimme Deffacts

As you work with CLIPS, you may become tired of typing in assertions. Although you cannot load assertions in from disk, there is a way to load in facts using the define facts keyword, **deffacts**. For example, type the following in response to a CLIPS prompt or enter it in a word processor and load it in from disk.

```
(deffacts walk
   (status walking)
   (walk-sign walk))
```

Following the (deffacts) keyword is the required name of this deffacts statement. Any valid symbolic atom can be used as the name. In this case, the name chosen was walk. After the name are the facts that will be asserted in the fact-list by this

deffacts statement.
The facts in a deffacts statement are asserted using the CLIPS **reset** command. Just enter

```
(reset)
```

in response to a CLIPS prompt and then give a (facts) command. You'll see the facts from the deffacts statement and a new fact generated by the (reset) command called

```
f-0     initial-fact
```

The (initial-fact) is automatically put in by a reset. Even without any deffact statements, a (reset) will always assert an (initial-fact). At first you may think this is pretty silly, but as you write more CLIPS programs you'll appreciate (initial-fact) more. The utility of (initial-fact) lies in starting programs running. A CLIPS program will not start running unless there are rules whose LHS are satisfied by facts. Rather than your having to type in some fact to start things off, the (reset) command asserts it for you as well as asserting the facts in deffacts statements.

The (reset) also has an advantage compared to a (clear) command in that (reset) doesn't get rid of all the rules. A (reset) leaves your rules intact. Like (clear), it also removes all activated rules from the agenda. Giving a (reset) command is a recommended way to start off program execution, especially if the program has been run before and the fact-list is cluttered up with old facts.

Selective Elimination

There may be times when you don't want a (deffacts) to assert facts when you issue a (reset) command. You can use the **undeffacts** command to prevent a (deffacts) from asserting facts. For example, enter

```
(undeffacts walk)
```

and then issue a (reset). If you check the fact-list, you'll see no facts from the (deffacts walk).

You can even get rid of initial-fact with undeffacts. Just enter the command

```
(undeffacts initial-fact)
```

Now, whenever you do a (reset), the initial-fact will not appear.
Besides facts, CLIPS also allows you to selectively eliminate rules by using the **excise** command. For example, to get rid of the take-a-walk rule, just specify the name of the rule as argument, as in

```
(excise take-a-walk)
```

Enter this rule in response to a CLIPS prompt and then issue a (rules) command. No rule called take-a-walk will be listed since take-a-walk has been excised.

A Matter Of Priority

Up to this point, we haven't seen any cases of competing rules in the agenda. But in most real situations, there is a need to specify which rules should have priority over other rules.
Let's consider the case of our robot trying to cross the street. There may be two rules in the agenda that are satisfied by the facts. One fact may be (green) while the other fact is that a police officer says to stop, even though the light is green. For example, enter the following two rules.

```
(defrule green-light
   (green)
=>
   (printout "Go" crlf))

(defrule police-officer
   (police-officer stop)
=>
   (printout "Stop" crlf))
```

Now enter the following

```
(deffacts conditions
   (green)
   (police-officer stop))
```

Do a (reset) and check the agenda. You'll see both rules are on the agenda since both are satisfied by the facts. When you (run), the "Stop" may appear before the "Go". However, since the rules have equal priority, you cannot count on the "Stop" always appearing first.

The correct thing to do is to assign a priority to the rules so that the "Stop" rule will always fire before the "Go" rule. CLIPS has a keyword called **salience** which is used to set the priority of rules.

The salience is set using a numeric value ranging from the smallest value of -10000 to the highest of 10000. If a rule has no salience explicitly assigned by the programmer, then CLIPS assumes a salience of 0. Notice that a salience of 0 is midway between the largest and smallest salience values. A salience of 0 does not mean that the rule has no salience but rather that it has an intermediate priority level.

Let's declare the salience of the police-officer rule to be higher than the green-light rule. We could select any values for the rules such that the value for police-officer is higher than green-light. A simple way of doing this is shown as follows.

```
(defrule police-officer
   (declare (salience 10))
   (police-officer stop)
=>
   (printout "Stop" crlf))
```

The (declare (salience)) statement assigns a salience value to a rule and must be the first pattern. In this case, the declared value of 10 for the police-officer rule is higher than the default value of 0 for the green-light rule. Although a salience value of 1 would also be higher, it's a good idea to spread out the salience values in case you later decide to add a rule with intermediate priority to the police-officer and green-light rules. That way, you won't have to modify the salience value of the police-officer rule since there are unused values of 1 to 9 that can be used for the intermediate rule.

Order, Order

Besides selecting competing rules in the agenda, salience has another important application. By assigning appropriate salience values, you can force rules to execute in a sequential fashion. Let's first look at an example in which salience is not used and then see how salience changes things.

```
(defrule first
   (first)
=>
   (printout "Rule 1" crlf))

(defrule second
   (second)
=>
   (printout "Rule 2" crlf))

(defrule third
   (third)
=>
   (printout "Rule 3" crlf))

(deffacts test
   (first)
   (second)
   (third))
```

Enter these rules, then do a (reset) and check the agenda. Now run the program and you'll see this output.

```
Rule 3
Rule 2
Rule 1
```

Notice the order of output statements. First Rule 3 is printed, then Rule 2, and finally Rule 1. The reason this occurs is that the activated rules are put on a stack. The first fact (first) activates the first rule, Rule 1. When the second fact is asserted, it activates the second rule, Rule 2, which is stacked on top of Rule 1. Finally, the third fact is asserted and its activated rule, Rule 3, is stacked on top of Rule 2.

In CLIPS, rules of equal salience which are activated by different conditional elements are prioritized based on the stack order of facts. Rules are fired from the agenda from the top of the stack down. So first Rule 3 is fired because it's on the top of the stack, then Rule 2 and finally Rule 1. If you change the order of the facts in the deffacts, you'll see that the last fact always produces the first rule to be fired and so forth. However, if rules are written which are all activated by the same conditional element, there is no guarantee of rule priority.

Now let's include salience. Suppose you want the rules to fire in the order Rule 1, Rule 2, and then Rule 3, no matter what the arrangement in deffacts. To accomplish this, just add salience as follows.

```
(defrule first
   (declare (salience 30))
   (first)
=>
   (printout "Rule 1" crlf))

(defrule second
   (declare (salience 20))
   (second)
=>
   (printout "Rule 2" crlf))

(defrule third
   (declare (salience 10))
   (third)
=>
   (printout "Rule 3" crlf))
```

Now do a reset and check the agenda. Notice how the salience values have rearranged the priority of rules in the agenda, as shown below.

```
30  first  f-1
20  second  f-2
10  third  f-3
```

When you run the program, you'll see that the order of rule firing is now Rule 1, Rule 2, and then Rule 3.

Watch It

CLIPS has several commands to help you debug programs. These commands allow you to **watch rules, watch facts,** and **watch**

activations on the agenda as the program is executing. To use these features, just enter the appropriate command for the information that you want to see. You can enter any or all of the commands.

```
(watch rules)
(watch facts)
(watch activations)
```

If you enter any one or all of these commands, CLIPS will print out the appropriate information on the screen.

To turn off the effect of these commands, enter the appropriate following command.

```
(unwatch rules)
(unwatch facts)
(unwatch activations)
```

Another useful command in debugging is the (run) command which takes an optional argument of the number of rule firings. For example, (run 21), would tell CLIPS to run the program and then stop after 21 rule firings. A (run 1) command allows you to step through a program 1 rule firing at a time.

The major difference between a rule-based system like CLIPS and languages such as Pascal or BASIC is that the rules of CLIPS can be activated in parallel while the statements of other languages are sequential in nature. That is, multiple rules can be put on the agenda.

In procedural languages like Ada, Pascal and BASIC, execution normally proceeds line by line in sequence. In CLIPS, any of the rules can be executed if certain conditions for the rules are met. So debugging an expert system program is not quite as straight-forward as debugging a program in a procedural language.

This is why the (watch) commands are so useful. Except for very simple programs, you can't just read a CLIPS program listing and figure out how it executes. Since the execution can occur from any triggered rule, you really need to know what's on the agenda to know what rule will be fired next. And to figure out why a rule got on the agenda, you have to know what the facts are.

Problems

1. Write a program to tell a robot what to do when it encounters a traffic light. The possible facts are

```
(light red)
(light green)
(light yellow)
(light blinking-red)
(light blinking-yellow)
(light blinking-green)
(light none)
(walk-sign walk)
(walk-sign dont-walk)
```

The status of the robot can be

```
(status walking)
(status driving)
```

2. Write a program to recognize and print the names of the following patterns. Store the patterns in separate deffacts. You'll have to include row numbers for some of the patterns because some rows are duplicates and CLIPS won't assert duplicate facts. Do not include row numbers in patterns that do not have duplicate patterns.

```
 1********   1     *       1            *
 2*      *   2    * *      2        *         *
 3*      *   3   *   *     3      *
 4*      *   4  *     *    4*
 5********   5*********    5       *
   Square      Triangle    6 *
                                 Big Dipper
1    *        1*****
2   * *       2*    *
3  *****      3*****
4 *     *     4*    *
5*       *    5*****
     A          B
```

Chapter 4 Variable Interests

The type of rules you've seen so far illustrate simple pattern matching of conditional elements to facts. In this chapter you'll learn very powerful ways to match and manipulate facts.

Let's Get Variable

Just as with other programming languages, CLIPS has **variables** available to store values. Variables give the programmer a powerful tool that is indispensible in many applications.

Variables in CLIPS are always written in the syntax of a question mark followed by a symbolic atom name. Some examples of variables are as follows.

```
?x
?sensor
?valve
?noun
?color
```

Be sure there is no space between the question mark and symbolic atom name or else CLIPS will misinterpret it.

Before a variable can be used, it should be assigned a value. As an example of a case where it's not assigned, enter the following rule which has variable ?x in the (printout).

```
(defrule test
   (initial-fact)
=>
   (printout ?x crlf))
```

In the above example, notice how initial-fact is used to trigger the rule. This is convenient since you can assert initial-fact by simply doing a (reset), which saves you the trouble of asserting initial-fact.

When you run the program, the output will be

```
Variable x not found
```

```
number
```

CLIPS gives an error message because it cannot find a value which has been **bound** to ?x. The terms bound and **bind** are used to describe the assignment of a value to a variable. Variables can be bound to symbolic atoms or numbers. Although no value was bound to ?x in the previous example, CLIPS returned a symbolic atom called "number". The atom "number" is meaningless and you should not rely on its assignment to unassigned variables in future releases of CLIPS.

Be Assertive

One common use of variables is to bind a value on the LHS and then assert the bound variable on the RHS. For example, enter

```
(defrule make-quack
   (duck ?sound)
=>
   (assert (?sound)))
```

Now assert (duck quack) and then (run) the program. Check the facts and you'll see that the rule has produced (quack) because the variable ?sound was bound to (quack).

Of course, you can also use a variable more than once. For example, enter the following. Also, be sure to do a (reset) and assert (duck quack) again.

```
(defrule make-quack
   (duck ?sound)
=>
   (assert (?sound ?sound)))
```

When the rule fires, it will produce (quack quack) since the variable ?sound is used twice.

What The Duck Said

Variables are also commonly used in (printout), as in

```
(defrule make-quack
   (duck ?sound)
=>
   (printout "The duck said " ?sound crlf))
```

Enter this rule and run to find out what the duck said. How would you modify the rule to put double quotes around quack in the output?

The Happy Bachelor

Retraction is very useful in expert systems and is usually done on the RHS. Before a fact can be retracted, it must be specified to CLIPS. To retract a fact from a rule, the **fact address** must first be bound to a variable on the LHS.

There is a big difference between binding a variable to the contents of a fact and binding a variable to the address of a fact. In the examples you've seen, such as (duck ?sound), a variable was bound to the contents of a fact. That is, ?sound was bound to quack. However, if you want to remove the fact whose contents are (duck quack), you must tell CLIPS the address of the fact to be retracted.

The address of the fact is specified using the **left arrow** operator, "<-". To make this operator, just type a "<" symbol followed by a "-".

As an example of fact retraction from a rule, enter the following example.

```
(defrule marriage
   ?bachelor <- (Rey)
=>
   (printout  "Rey is now happily married" crlf)
   (retract ?bachelor))
```

Assert the fact (Rey) and then run. After execution, check the fact-list and you'll see no (Rey) since it has been retracted. The conditional element in the rule assigns the address of (Rey) to the variable ?bachelor. The (retract) then removes the fact (Rey) whose address was assigned to ?bachelor.

Variables can be used to pick up a fact content at the same time as an address. For example

```
(defrule marriage
   ?bachelor <- (?name)
=>
   (printout  ?name " is now happily married" crlf)
   (retract ?bachelor))
```

The above rule will fire when ?name is assigned to any fact. Enter the rule and then assert the facts (duck), (Marlon), and (Gary). When you run, you'll see the three happily married ex-bachelors. Notice how the rule fired on all facts which matched the conditional element (?name).

John Smith's Marriage

Will the marriage rule fire on a fact with two symbolic atoms like (John Smith)? Try it and see. The rule will not fire because there are two atoms in (John Smith) while the conditional element specifies one atom to be matched to (?name).

So how do we get John Smith married? One approach you might try is the following, in which two variables called ?name are specified to match the two atoms in (John Smith).

```
(defrule marriage
   ?bachelor <- (?name ?name)
=>
   (printout  ?name ?name" is now happily married" crlf)
   (retract ?bachelor))
```

However, this rule won't fire on (John Smith) because the conditional element (?name ?name) specifies that the two atoms must be the same since the same variable ?name is used for both atoms.

The way to get John Smith married is to use two different variables, as follows.

```
(defrule marriage
   ?bachelor <- (?first ?last)
=>
   (printout ?first ?last " is now happily married" crlf)
   (retract ?bachelor))
```

Enter this rule and then assert (John Smith). When you run,

you'll see that John Smith finally gets married.
Suppose John has a brother whose first name is Smith (their parents liked simple names). Will this rule make Smith Smith get married too? Assert (Smith Smith) and run again. You'll see that there will be a double wedding indeed.
Notice that the conditional element (?first ?last) matched any two symbolic atoms while (?name ?name) required that the two atoms were the same.

How To Make John Very, Very Happy

The (assert) can be used with (retract) to produce some interesting effects. As a simple example, suppose we wanted to make John not just happy, but very, very happy. The following program shows how John can become very, very happy. Enter it and assert (John Smith). Be sure to save the program before you run it.

```
(defrule ecstatic
   ?bachelor <- (?first ?last)
=>
   (printout ?first " " ?last " is a happy man" crlf)
   (retract ?bachelor)
   (assert (?first ?last)))
```

The program executes an infinite loop since it asserts a new (John Smith) after retracting the old ?bachelor. Note that to make a new (John Smith) you must assert the individual atoms such as ?first and ?second. CLIPS is not designed to allow assertions of entire facts with statements like (assert (?bachelor)). After the new (John Smith) is asserted, the rule then fires on the new (John Smith).
The only way to stop an infinite loop like this is by interrupting CLIPS. Use a Control C or appropriate interrupt command for your computer. You'll have to restart CLIPS again.
Will the program still produce an infinite loop if you remove the (retract) statement? The answer is no since the new attempted assertions are duplicates of the fact in memory.

Who's Eligible

Rather than having to specify a certain fact to trigger a rule, you can just specify a general conditional element using a **wildcard.** For example, suppose you're running a dating service and a woman specifies that she only dates men whose first name is John. There are really two criteria in this specification since there is an implication that the man must have more than one name. So a plain (John) will not do because there is only one name in the fact.

This type of situation in which only part of the fact is specified is very common and very important. To solve this problem, a wildcard can be used to fire a rule to print out the John's.

The simplest form of wildcard is called a **single-field wildcard** and is shown by a question mark, "?". A single-field wildcard stands for one atom. The following rule and deffacts show how a wildcard can be used. Enter and run.

```
(defrule eligible
   (John ?)
=>
   (printout "There is an eligible John" crlf))

(deffacts bachelors
   (Tom)
   (John)
   (John Smith)
   (John Henry)
   (John Henry Smith))
```

The conditional element includes a wildcard to indicate that John's last name is not important. So long as his first name is John, the rule will be satisfied and fire. Because the wildcard is in the second field, that is, the position of the second atom of a fact, only facts of two atoms can possibly satisfy the rule. In other words, only John's with exactly two names need apply.

Suppose you want to specify John's with exactly three names? All you'd have to do is write a conditional element like

```
(John ? ?)
```

or if only persons with three names whose middle name was John

```
(? John ?)
```

or if only the last name was John

```
(? ? John)
```

Another interesting possibility is if the the John must be the first name, but only those with two or three names are acceptable. One way of solving this problem is to write two rules. For example

```
(defrule eligible
   (John ?)
=>
   (printout "There is an eligible John" crlf))

(defrule eligible-three-names
   (John ? ?)
=>
   (printout "There is an eligible John" crlf))
```

Enter and run this and you'll see that both the John's with two and three names are printed.

Going Wild

Rather than writing separate rules to handle each field, it's much easier to use the **multi-field wildcard.** The multi-field wildcard is a dollar sign followed by a question mark, "$?", and represents zero or more occurences of an atom. The two rules for eligibility can now be written in a single rule as follows. Also shown is the deffacts of eligible bachelors.

```
(defrule eligible
   (John $?)
=>
   (printout "There is an eligible John" crlf))

(deffacts bachelors
   (Tom)
   (John)
   (John Smith)
   (John Henry)
   (John Henry Smith))
```

When you enter and run, you'll see that the rule fires four times since there are four (John) facts.

Wildcards have another important use because they can be attached to a symbolic atom to create a variable such as ?x, $?x, ?name, $?name. You've already seen examples of single-field variables at the beginning of this chapter. Now you can understand why the variables had a "?" in front. The attached variable will be equated to either a single-field wildcard if only the "?" is used or a multi-field wildcard, "$?", if that is used.

As example of a multi-field variable, the following version of the rule also prints out the name of the matching fact because a variable is equated to the name that matches.

```
(defrule eligible
   (John $?name)
=>
   (printout "There is an eligible John " $?name crlf))
```

When you enter and run, you'll see the names of all the eligible John's. The multi-field wildcard takes care of any number of atoms. No matter how long John's other names are, the multi-field wildcard will take care of it.

What do you think the matches would be if the conditional element was ($?name) alone? Try it and you'll see that the rule matches to every fact, even (Tom).

Suppose you wanted a match of all people who had a John somewhere in their name, not necessarily as their first name. Notice that this requirement is not the same as in the previous rule. Some possible matches would be

```
(John)
(Tom John)
(Tom Henry John)
(Tom John Henry)
(Tom Henry John Smith)
```

The following version of the rule would match all facts with a John in them and print out the names.

```
(defrule eligible
   ($?first John $?last)
=>
   (printout "There is an eligible " $?first " John " $?last crlf))
```

Enter and run this rule and you'll see all the matching John's. The conditional element matches any names that have a John anywhere in them.

Single and multi-field wildcards can be combined. For example, the conditional element

```
(? $? John ?)
```

means that the first and last names can be anything, and that the name just before the last must be John. For example,

Name	Match
(John)	No
(John Smith)	No
(Tom John Smith)	Yes
(Tom John Henry Smith)	No
(Tom Henry John Smith)	Yes

Note that the "$?" can match zero or more occurences of an atom. Matching zero occurences means that there can be a match with no atom. So a conditional element like ($?name) will match any fact. In contrast, the "?" requires an atom to match.

Problems

1. A factory has different bars in stock. The ID and dimensions of the stock are given as follows.

ID	Length	Width	Height
Fe-1	10	10	10
Cu-2	20	10	10
Cu-3	20	10	20
Pt-4	10	20	10
Fe-5	30	30	10
Fe-6	30	20	10
Fe-7	10	30	20
Cu-8	10	20	30
Pt-9	30	20	10
Pt-10	20	20	20

Table 4.1

Write a program that will classify the bar stock according to the following definitions.

square cross section - length=width
square side - width=height
cubes - length=width=height

The program should print out the stock ID and all the dimensions of stock that matches the definitions. Objects may be in more than one category. For example, a cube will also be listed as having square cross section and square side.
The priority of printing should be all the cubes, then all square cross section, and then all those with square sides. Finally the program should print out the complete inventory exactly as shown in Table 4.1.

2. Write a program to print all permutations of the facts (John), (Henry), and (Smith). That is the program should print all the possible combinations of the three facts, such as

```
John Henry Smith
John Smith Henry
Henry John Smith
```

and so forth.

Chapter 5 A Little Math

Besides dealing with symbolic facts, CLIPS can also perform calculations. Although CLIPS was not designed for efficiency in number crunching, it can do a wide variety of math operations. You'll find the computational capability of CLIPS useful in many applications.

Two Packages

CLIPS is available in different packages so that you can customize it to fit your needs. The standard CLIPS **kernal** provides only the basic arithmetic operations such as addition, subtraction, multiplication and division.

The CLIPS source disk has both the kernal as well as a math library of C routines. You can link and compile this library with the kernal to produce an enhanced CLIPS with a very powerful library of math functions. The math library is included as an example of how users can enhance CLIPS to suit their needs. The disadvantage of the math library version of CLIPS is that it adds to the size of CLIPS. The amount of extra memory required will depend on your compiler.

An expert system language like CLIPS is not designed to produce application programs that do a lot of calculations. Although the math functions of CLIPS are very powerful, they are primarily meant for modification of numbers that are being reasoned about by the application program. Other languages such as FORTRAN are better for number crunching in which little or no reasoning about the numbers is done.

Let's Be Precise

Since CLIPS is designed for portability, it can be put on many different models of computers. However, different models usually treat numbers in different forms. You should work through the examples in this chapter carefully on your computer and note any differences.

In particular, any rules which depend on a numeric value in a conditional element may lead to problems in portability of CLIPS code from one model of computer to another. All of the examples in this book were done on an IBM PC XT and should be fairly close to what you'll see on other computers using single-precision, floating point arithmetic.

As an example of the way CLIPS displays numbers, enter the following (deffacts) or just (assert) the numbers.

```
(deffacts math
   (0)
   (1)
   (-5)
   (1.5)
   (-.000123456)
   (123456)
   (1.33e7)
   (2.5e+4)
   (-6.924e-10)
   (1e50)
   (1e-50))
```

After you do a (reset), check (facts) and you'll see facts like the following. Brief comments have been inserted to explain the facts while a more detailed discussion follows.

```
f-1     (0)
f-2     (1)   ;really floating point, just appearance of integer
f-3     (-5)  ;see note for f-2
f-4     (1.5) ;unchanged appearance from asserted fact
f-5     (-0.00012346) ;note round-off to five significant digits
f-6     (1.2346E+05)  ;see note for f-5,also exponential notation
f-7     (1.3300E+07)  ;false precision displayed
f-8     (25000)       ;see note for f-2,also now fixed notation
f-9     (-6.9240E-10) ;see note for f-7
f-10    (3.4028E+38)  ;this fact and
f-11    (0)           ;this one are not correct because they
exceed the numeric limits of the compiler used for CLIPS.
Different compilers will have different limits.
```

Even the appearance of facts can be deceptive if the numbers exceed the range allowed by the compiler you used for CLIPS. For example, on some compilers, the asserted fact (1e-50) might be really stored as 0 if it's smaller than the smallest number allowed by the compiler. On other compilers, it might really be stored as 1e-50.

Numbers whose absolute magnitude are smaller than 0.0001 or greater than or equal to 100000 will be expressed in exponential notation. Numbers with absolute magnitude from 0.0001 to less than 100000 will be expressed in fixed form, even if entered in exponential notation.

The **precision** of a number refers to how many **significant digits** it has. For example, the numbers 1.23e-3 and 0.000123 both have three significant digits. Leading zeroes do not count in determining the number of significant digits.

The **accuracy** of a number refers to the truth of the number. For example, suppose the height of a chair is measured with a ruler which has divisions to a millimeter as 0.712 meters. Then suppose the same chair is measured with another ruler which has rulings of only tenths of a meter as 0.4 meters. Notice that the measurements are quite different. Which is correct? At first you might say that the 0.712 reading is correct since it has more significant digits. But that is not necessarily so. It could be that the first ruler was incorrectly calibrated. While 0.712 is certainly more precise it may not be more accurate.

The precision and range of numbers in CLIPS will depend on the compiler that you use to compile it. CLIPS will normally show five significant digits and this should be the same in all implementations. However, the internal precision in which numbers are represented may vary from one compiler to another and this can affect the triggering of rules.

Sometimes a false precision can be shown, as in the case of f-7. Note that the asserted fact (1.33e7) has only three significant digits and yet CLIPS shows five significant digits. Although leading zeroes are not significant, trailing zeroes are significant. So 1.3300E+07 is shown as if it was known more precisely. All we really know is that the original asserted fact of 1.33e7 was within the range 1.334e7 to 1.330e7. A number greater than 1.334e7 would have been expressed as 1.35e7 while a number smaller than 1.330e7 would have been written as 1.29e7.

The appearance of false precision arises because CLIPS is designed to always show five significant digits. However, the false precision will not hurt your calculations since it is just the displayed appearance. It's only important that when you interpret the final results that you realize they cannot be more precise than the most imprecise number.

As an example of how the precision of numbers affects rule firing, enter the following.

```
(defrule num-check
   (1.5)
=>
   (printout "valid number" crlf))
```

When you (assert (1.5)) the rule will fire. Now retract the fact and assert 1.5000001. On an IBM PC where CLIPS was compiled with the Lattice C Compiler, the fact was stored as (1.50000012) as shown by a (facts). In this case the rule would not fire. However, if the fact 1.50000001 was asserted, the fact was stored as 1.5 and so the rule did fire.

Depending on your implementation of CLIPS, you may need more zeroes to make the rule fire. But eventually, the number that you assert will be less than the precision of CLIPS and the rule will fire. So if you design a program with conditional elements which include numbers, you should consider the precision and range of numbers that will cause erroneous rule firings. Conditional elements which depend on numbers should be examined very carefully, especially if you plan to run the program on a different model of computer later.

If you need a lot of precision with decimal numbers, be careful since most compilers produce code using binary number representation. This can lead to inaccurracy and imprecision. For more details on binary numbers and imprecision, see BASIC: Advanced Concepts by J. Giarratano, published by Howard W. Sams, Inc.

It's Elementary

CLIPS provides the elementary **arithmetic operators** as shown in the following table.

CLIPS Operators

+	addition
-	subtraction
*	multiplication
/	division
**	exponentiation

All arithmetic and numbers are in floating point.

Numeric expressions are represented in CLIPS according to the style of LISP. In LISP and CLIPS, a numeric expression that you would customarily write as 2 + 3 must be written in the LISP and CLIPS **prefix form,** (+ 2 3). The customary way of writing numeric expressions is called **infix form** because the math operators are in between the **operands** or **arguments.** In the prefix form of CLIPS, the operator must go before the operands and parentheses must surround the numeric expression.

One way of evaluating a numeric expression in CLIPS is using the = **comparison function** in an (assert) action. The general form of an arithmetic calculation in CLIPS with two arguments is

```
(operator arg arg)
```

where "operator" can be any of the arithmetic operators and "arg" stands for the argument used by the operator.

To see how these arithmetic operations perform on your computer, enter the following rules, then assert (numbers 2 3) and run. Also shown by each rule is a comment describing what each rule does. Note that the comment is in infix notation for your information only. Infix cannot be used in a rule.

```
(defrule addition
   (numbers ?x ?y)
=>
   (assert (=(+ ?x ?y))))    ; ?x + ?y

(defrule subtraction
   (numbers ?x ?y)
=>
   (assert (=(- ?x ?y))))    ; ?x - ?y

(defrule multiplication
   (numbers ?x ?y)
=>
   (assert (=(* ?x ?y))))    ; ?x * ?y

(defrule division
   (numbers ?x ?y)
=>
   (assert (=(/ ?x ?y))))    ; ?x / ?y

(defrule exponentiation
   (numbers ?x ?y)
=>
```

```
   (assert (=(** ?x ?y))))   ; ?x ** ?y
```

After you run, check the facts. The answer for division will probably show a round-off error in the last digit.

At the present time, CLIPS is not designed to evaluate numeric expressions at the top-level. So you cannot assert a top-level command like (assert (=(+ 2 3))). Numeric evaluation must be done within a rule.

In a rule, you can assert other atoms along with a numeric expression. For example, the addition rule could be

```
(defrule addition
   (numbers ?x ?y)
=>
   (assert (sum =(+ ?x ?y))))
```

Be sure to put a space between the atom "sum" and the following "=". Remember that a space must separate atoms in a fact. The "=" tells CLIPS to evaluate the following fact before asserting it. So "=(+ ?x ?y)" is evaluated as 2 + 3 = 5 and the resulting atom of 5 is used for the assertion (sum 5). Run this version of the rule and you'll see the fact (sum 5).

Extensive Arguments

The arguments in a numeric expression can be extended beyond two for all operators except exponentiation. The same sequence of arithmetic calculations is performed for more than two arguments. The following example shows how three arguments are used. Evaluation proceeds from left to right.

```
(defrule addition
   (numbers ?x ?y ?z)
=>
   (assert (=(+ ?x ?y ?z))))   ; ?x + ?y + ?z

(defrule subtraction
   (numbers ?x ?y ?z)
=>
   (assert (=(- ?x ?y ?z))))   ; ?x - ?y - ?z

(defrule multiplication
   (numbers ?x ?y ?z)
=>
   (assert (=(* ?x ?y ?z)))   ; ?x * ?y * ?z

(defrule division
   (numbers ?x ?y ?z)
=>
   (assert (=(/ ?x ?y ?z)))   ; ?x / ?y / ?z
```

Enter the above program and assert (numbers 2 3 4). After you run the program, you'll see the following facts. Note that the division fact, f-5, may be different on your computer because of the compiler you used for CLIPS.

```
f-2      (9)
f-3      (-5)
f-4      (24)
f-5      (0.16666667)
```

The infix equivalent of a multiple argument CLIPS expression can be expressed in the following general way.

arg operator arg operator arg operator arg ...

where the dots, "...", symbolize the pattern repeated. Another way of symbolizing this is

arg [operator arg]

where the square brackets mean that there can be multiple terms. Note that you cannot write a CLIPS expression with dots or square brackets. The dots and brackets are just a shorthand notation.

Mixed Results

Mixed calculations can also be done in the prefix notation. For example, suppose you want to evaluate the infix expression

```
?x + ?y * ?z
```

One important fact about CLIPS and LISP calculations is that there is no built-in precedence of arithmetic operations. That is, in other computer languages, multiplication and division rank higher than addition and subtraction and the computer does the higher ranked operations first. However, in LISP and CLIPS, there is no built-in precedence. Everything is simply evaluated from left to right with parentheses determining precedence.

In the example of ?x + ?y * ?z, the customary way to evaluate it is to multiply ?y by ?z and then add the result to ?x. However, in CLIPS, you must explicitly write the precedence. The example would be written in the following rule.

```
(defrule mixed-calcl
   (numbers ?x ?y ?z)
=>
   (assert (result =(+ ?x (* ?y ?z)))))  ; ?x * ?y + ?z
```

In this rule, the innermost parentheses is evaluated first and so ?y is multiplied by ?z. The result is then added to ?x. Also, note that you don't need an "=" function until the final atom is to be asserted. That is, you don't need to put an "=" in front of every operation in parentheses.

If you wanted ?x + ?y * ?z, where again multiplication is done first, the rule would be

```
(defrule mixed-calc2
   (numbers ?x ?y ?z)
=>
   (assert (result =(+ (* ?x ?y) ?z))))  ; ?x * ?y + ?z
```

Enter and run these rules for the (numbers 2 3 4) and see if the results are as expected.

Binding Things Up

In order to print out the value of a math calculation, the result must first be bound to a variable using the **bind** function. The bound variable can then be printed out. For example,

```
(defrule addition
   (numbers ?x ?y)
=>
   (assert (answer (=(+ ?x ?y))))
   (bind ?answer (+ ?x ?y))
   (printout "answer is " ?answer crlf))
```

Enter and run this and you'll see that the answer is printed out as well as asserted.

Notice the difference in syntax between the (assert) with its "=" function and the bind which does not use the "=". The bind function does not use the "=" because CLIPS assumes that whatever follows the variable argument in a bind should be evaluated. In contrast, when CLIPS encounters an (assert), it does not know at first if an argument is to be evaluated. So an "=" is necessary to tell CLIPS that the argument in the (assert) should be evaluated.

What's At The Root

The math functions can return a value if they are bound to a variable using the bind function. The following example shows how the **square root function** is first bound and then printed out.

```
(defrule square-root
   (number ?value)
=>
   (bind ?answer (sqrt ?value))
   (printout "square root of " ?value " is " ?answer crlf))
```

To run, first assert a fact such as (number 4).

The bind function can be used to bind any atom to a variable. For example,

```
(defrule bind-test
   (number ?value)
=>
   (bind ?answer ?value)
   (printout "answer is " ?answer crlf))
```

will bind ?answer to ?value. Try running this after asserting (number 4) and you'll see the answer of 4.

Math functions can be included in arithmetic expressions using prefix notation as shown following.

```
(defrule square-root
   (number ?value)
=>
   (bind ?answer (+ 4 (sqrt ?value)))
   (printout "square root of " ?value " is " ?answer crlf))
```

If you assert (number 4) and run, you'll see the answer of 6 printed out from ?answer. Math functions are treated just like any other argument in a numeric expression, except that they require parentheses around them.

Math Functions

Following is a list of the math functions in the math library. To access these functions, you must first link the library to the CLIPS kernal and compile it to produce the math version of CLIPS. The math version will execute just like CLIPS with the added enhancement that you can access all of the following math functions.

Note: all trig functions are expressed in radians.
      <fun> stands for a function.
      <arg> stands for the one argument of the function.

Trig And Hyperbolic Functions

Format : (<fun> <arg>)

Example:

```
(defrule trigtest
   (number ?value)
=>
   (bind ?z (sin ?value)))

(assert (number 1))
```

In the above example, ?z will be bound to the sine of 1.

Standard Trigonometric Functions

sin
cos
tan

sec
csc
cot

Inverse Trigonometric Functions

acos
asin
atan
asec
acsc
acot

Hyperbolic Functions

cosh
sinh
tanh
sech
csch
coth

Inverse Hyperbolic Functions

acosh
asinh
atanh
asech
acsch
acoth

Maximum And Minimum Functions

The following functions return the minimum and maximum of a set of numbers. The double angles, "<<" and ">>", around the arguments indicate one or more arguments are allowed.

Format : (<fun> <<arg>>)

min : returns the minimum argument

max : returns the maximum argument

Example:

```
(defrule test
   (numbers ?x ?y)
=>
   (bind ?z (min ?x ?y)
   (bind ?w (max ?x ?y)))

(assert (numbers 2 3))
```

In the above example, ?z will be bound to 2.
?w will be bound to 3.

Modulus Function

The modulus function returns the modulus of the first argument by the second argument. The modulus is defined as the integer result of division of the first argument by the second. Since CLIPS deals only with floating point numbers, the modulus is returned as a floating point number instead of the standard integer.

Format : (mod <arg> <arg>)

Example:

```
(defrule modtest
   (numbers ?x ?y)
=>
   (bind ?z (mod ?x ?y))

(assert (numbers 3 2))
```

In the above example, ?z is bound to 1.

Log Functions

log : returns the log to base e

log10 : returns the log to base 10

Format : (<fun> <arg>)

```
Example:      (defrule logtest
                 (number ?value)
              =>
                 (bind ?z (log ?value))
                 (bind ?w (log10 ?value)))

              (assert (number 5))
```

In the above example, ?z will be bound to the log of 5 while ?w will be bound to the log to base 10 of 5.

Exponential Function

exp : returns e raised to the power of the argument, that is, e**arg

Format : (exp <arg>)

```
Example:      (defrule exptest
                 (number ?value)
              =>
                 (bind ?z (exp ?value)))

              (assert (number 1))
```

In the above rule, ?z is bound to exp of 1.

Square Root

sqrt : returns the square root of the argument

Format : (sqrt <arg>)

```
Example:      (defrule sqrtest
                 (number ?value)
              =>
                 (bind ?z (sqr ?value)))

              (assert (number 4))
```

In the above example, ?z is bound to the square root of 4.

Problems

1. Check the limits of accuracy on your computer. Assert

```
(number 1e-38)
(number 1e-50)
(number 1e-70)
```

and see how often the rule

```
(defrule test-limit
   (number 0)
=>
   (printout "Rule fires" crlf))
```

fires. Use (watch rules) and (watch facts) commands to see which facts are triggering the rules. Assert other numbers to narrow down the limit of the largest number that will be interpreted as 0.

2. Write a program to calculate the area of various geometric shapes. If the user asserts one of the following facts, the program will calculate and print out the area. If something is asserted that is not one of the shapes shown, the program should print out an error message. Run for the examples shown. For Pi, assert (Pi 3.14159).

Examples	Meaning	Area
(square 10)	(square side)	S * S
(triangle 6 20)	(triangle base height)	1 / 2 * B * H
(rectangle 5 3)	(rectangle length width)	L * W
(circle 3)	(circle radius)	Pi * R * R

Chapter 6 How To Be In Control

Up to this point, you've been learning the basic syntax of CLIPS. Now you'll see how to apply the syntax you've learned to more powerful and complex programs. You'll also learn some new syntax for keyboard input, and see how to compare values and generate loops.

Let's Start Reading

Besides matching a conditional element, there is another way that a rule can get information. CLIPS can read information you type from the keyboard using the **read function.**

The following example shows how (read) is used to calculate the square of a number. Note that no extra (crlf) is needed after the (read) to put the cursor on a new line. The (read) automatically resets the cursor to a new line.

```
(defrule square-number
   (initial-fact)
=>
   (printout "number? " crlf)
   (bind ?num (read))
   (bind ?square (* ?num ?num))
   (printout "square of " ?num " is " ?square crlf))
```

Since the rule is designed to use keyboard input on the RHS, it's convenient to trigger the rule with (initial-fact). Otherwise, you'd have to make up some dummy fact to trigger the rule.

The (read) function is not a general purpose function that will read anything you type on the keyboard. One limitation is that (read) will only read one atom. So if you try to read

```
this is a duck
```

only the first atom, "this" will be read. To (read) all of the input. you must enclose the input within double quotes. Of course, once the input is within double quotes, it is a single literal atom. You can't access the individual words "this", "is", "a", and "duck" with CLIPS functions since an atom can't be split up in CLIPS. To split up an atom, you must create and call an external function.

The second limitation of (read) is that you can't input parentheses unless they are within double quotes. Just as you can't assert a fact with parentheses, you can't (read) parentheses.

Even if (read) could read multiple atoms, there would be a limitation. Because (bind) can only bind one variable at a time, you can't read multiple variables in a single (bind). To read multiple values and bind them, you'll have to use multiple (bind) actions which each contain a (read). Also, since CLIPS does not have a concatenation operator, you cannot combine separate facts.

Reading Your Assertions

You can read an atom into an (assert) action. For example, suppose you wanted to modify the square program to give the user a choice of running the program again. This is a useful feature in many programs because otherwise the program ends after one calculation.

The following version of the square program shows how to run the program over again by retracting and then asserting a dummy fact called start-fact again in the do-another rule. Although it is possible to use initial-fact instead of start-fact, it is poor programming because other rules may depend on initial-fact to trigger them. As long as your programs don't retract (initial-fact), it's safe to use it. Otherwise, use your own fact. For convenience, you may wish to keep a (deffacts) of (start-fact) in a file and load it in after your rules.

```
(defrule square-number
   (start-fact)
=>
   (printout "number? " crlf)
   (bind ?num (read))
   (bind ?square (* ?num ?num))
   (printout "square of " ?num " is " ?square crlf)
   (printout "Do another calculation? (y or n) " crlf)
   (assert (response =(read))))
```

```
(defrule do-another
   (response y)
   ?initial <- (start-fact)
=>
   (retract ?initial)
   (assert (start-fact)))
```

Sample output of the program is as follows. Try some other numbers, especially very large and very small numbers to see how your computer handles their precision.

```
number?
2
square of 2 is 4
Do another calculation? (y or n)
y
number?
4
square of 4 is 16
Do another calculation? (y or n)
```

Notice how the two rules work together. This is a common method in expert systems to generate a loop where one rule triggers another. Since the flow of control in a rule-based system is not sequential, you must program the appropriate interactions between rules.

Control Your Loop

There are many cases in which it's useful to repeat a calculation or other information processing. The common way of doing this is by setting up a loop. In the previous example, you've seen how a loop is set up until the user responds no to a question. But there are also many situations in which you want a loop to terminate automatically as the result of some test. In the previous example, the test was asking the user whether they wanted to do another calculation. In a more general sense, the test of loop termination will be a comparison of values.

The **test** function provides a very powerful way to compare numbers, variables, and strings on the LHS. The basic syntax of test is

```
(test (<fun> <<arg>>))
```

where <fun> is a **predefined function** of CLIPS and <<arg>> stands for one or more arguments required by the function. The (test) is used as a conditional element on the LHS. So a rule will only be triggered if the (test) is satisfied.

As a simple example of a (test), consider the problem of writing a program to generate the squares of numbers up to a maximum number. The following program uses (test) in a rule to decide when to stop printing the squares.

```
(defrule input-max
   (start-fact)
=>
   (printout "number of loops? " crlf)
   (bind ?max (read))
   (assert (loop 0 ?max)))    ;initialize ?count to 0

(defrule print-squares
   (loop ?count ?max)
   (test (<= ?count ?max))    ;test if ?count is <= ?max
=>
   (bind ?square (* ?count ?count))
   (printout "square of " ?count " is " ?square crlf)
   (assert (loop =(+ ?count 1) ?max))) ;?count = ?count + 1
```

The fact (loop) contains information about how many numbers have been printed in ?count, and the maximum number, ?max, to be printed. Notice that the maximum number must be included in a fact and input to print-squares for every iteration of the loop. While the program works, it is wasteful of memory. How would you change it to reduce all the (loop) facts generated?

The (test) checks if ?count is less than or equal to ?max using the **less than or equal function, "<="**. On the last iteration of the loop, ?count = ?max and so the rule will not be triggered again. The arguments of "<=" are ?count and ?max. Look again at the syntax of (test) described before, and you can now see how (test (<= ?count ?max)) matches (test (<fun> <<arg>>).

In general, you can understand how the predefined function acts on what follows by thinking of it following the first argument. For example,

```
(<= ?count ?max)
```

in prefix form is like saying

(?count <= ?max)

in customary infix.

There are many predefined function provided for you by CLIPS. The opposite of the predefined function is the **external function** or **user-defined function.** An external function is a function that you write in C and link to CLIPS. For more information, see the CLIPS Reference Manual.

The following table shows the predefined functions.

Symbol	Predefined Function
	Logical Predefined Functions
!	not(inverse) function
&&	and function
||	or function
	Comparison Predefined Functions
=	equal(numeric) function
eq	equal(strings or numbers) function
!=	not equal function
>=	greater than or equal to function
>	greater than function
<=	less than or equal to function
<	less than function
	Arithmetic Predefined Functions
/	division function
*	multiplication function
**	exponentiation function
+	addition function
-	subtraction function

All of the Comparison Functions except "eq" will give an error message if used to compare a number and non-numbers, called **strings.** The "eq" function should be used for checking items whose types are not known in advance.

The (test) can be used for more complex arguments. However, you'll have to get used to the prefix notation to use (test) effectively.

For example, suppose you want to see if two points have a positive slope. In the customary infix way, we can write this as

```
(y2 - y1) / (x2 - x1)   > 0
```

In order to write this in prefix form, let's start off by thinking of the numerator as (Y) and the denominator as (X). So we can write the above as

```
(Y) / (X)   > 0
```

The prefix form for division is then

```
(/ (Y) (X))
```

since in prefix the operator comes before the argument. Now we want to see if the division is greater than 0. So the prefix form is used with (test) as follows.

```
(test (> (/ (Y) (X)) 0))
```

In infix form, Y = y2 - y1. But we want the prefix form since we're making a prefix expression. So (- y2 y1) will be used for (Y) and (- x2 x1) for (X). Now just replace (Y) and (X) by their prefix forms to get the final expression of whether the two points have a positive slope.

```
(test (> (/ (- y2 y1) (- x2 x1)) 0))
```

As you become more proficient in prefix form, you'll be able to write the prefix form automatically. At first, it's probably best if you write down the steps in converting from infix to prefix until you get the hang of it.

Let's Be Logical

The logical functions of (test) are very useful in expressing multiple relationships. For example, suppose you wanted to check for valid entry of numbers, such as dates. If the month is entered as a number, it must be greater than or equal to 1 and less than or equal to 12. The "and" function "&&" can be used to express this relationship in the test

```
(test (&& (>= ?month 1) (<= ?month 12)))
```

where ?month would contain the month number. The "&&" is formed by pressing the "&" key twice.
Likewise an invalid month number would be one that is less than 1 or greater than 12. This can be expressed using the logical "or" function, "||", formed by pressing the "|" key twice.

```
(test (|| (< ?month 1) (> ?month 12)
```

A program to tell whether a month is valid or invalid is shown following. Enter and run it for some different month numbers.

```
(defrule ask-month-number
   (start-fact)
=>
   (printout "Number of month? " crlf)
   (bind ?number (read))
   (assert (month ?number)))

(defrule valid-month
   ?start-fact <- (start-fact)
   ?month <- (month ?number)
   (test (&& (>= ?number 1) (<= ?number 12)))
=>
   (printout "Valid month" crlf)
   (retract ?start-fact ?month)
   (assert (start-fact)))

(defrule invalid-month
   ?start-fact <- (start-fact)
   ?month <- (month ?number)
   (test (|| (< ?number 1) (> ?number 12)))
=>
   (printout "Invalid month" crlf)
   (retract ?start-fact ?month)
   (assert (start-fact)))
```

Note that after facts are no longer needed, they are retracted. This prevents a lot of facts from building up in memory and slowing down execution because CLIPS tries to match them to rules.

Name That Month

While it's nice to know if a month is valid, it would be even nicer if the program would name the month. Following is a modification of the program which does this. Enter and run for some month numbers and you'll see the month names printed out.

```
(defrule ask-month-number
   (start-fact)
=>
   (printout "Number of month? " crlf)
   (bind ?number (read))
   (assert (month ?number)))

(defrule valid-month
   ?month <- (month ?number)
   (test (&& (>= ?number 1) (<= ?number 12)))
=>
   (printout "Valid month" crlf)
   (retract ?month)
   (assert (valid-month ?number)))

(defrule invalid-month
   ?start-fact <- (start-fact)
   ?month <- (month ?number)
   (test (|| (< ?number 1) (> ?number 12)))
=>
   (printout "Invalid month" crlf)
   (retract ?start-fact ?month)
   (assert (start-fact)))

(defrule print-month
   ?valid <- (valid-month ?number)
   (name ?month ?number)
   ?start-fact <- (start-fact)
=>
   (printout "Month is " ?month crlf)
   (retract ?start-fact ?valid)
   (assert (start-fact)))
```

```
(deffacts month-names
   (name January 1)
   (name February 2)
   (name March 3)
   (name April 4)
   (name May 5)
   (name June 6)
   (name July 7)
   (name August 8)
   (name September 9)
   (name October 10)
   (name November 11)
   (name December 12))
```

The data for the month names and numbers are stored in the facts of the (deffacts) statement. The rule print-month is triggered when there is a fact (valid-month ?number) made by the rule valid-month. Notice how the rule valid-month was modified to assert a fact (valid-month ?number) which is a conditional element of the rule print-month. The second conditional element is the fact which matches the month name and number. The third conditional element is just to match the fact start-fact so that the program does not end.

Let's talk a little now about efficiency. A minor point, but one worth mentioning, is the choice of names in facts. Consider the difference between (valid-month ?number) and an alternative (valid month ?number). The (valid-month ?number) pattern tries to match two atoms, "valid-month" and "?number". In contrast, the pattern (valid month ?number) must try to match three atoms to the patterns, "valid", "month", and "?number".

So if the pattern (valid month ?number) had been used, CLIPS would have to do more work in matching, which decreases its efficiency. Although this choice of names is not a big problem in this program, it can become aggravated if you use a lot of patterns unnecessarily.

Another factor that affects efficiency is the order in which conditional elements are listed affects the efficiency of a program. If the order was reversed like this

```
(name ?month ?number)
   ?valid <- (valid month ?number)
   ?start-fact <- (start-fact)
```

then CLIPS would waste time in checking to see if the rule should be put on the agenda.

The reason for this inefficiency is that CLIPS would always find 12 matches to the first conditional element. Then CLIPS would go on to the second conditional element and find one match of the valid-month number. Finally, it would find 1 match for the (start-fact). So a total of 12 x 1 x 1 = 12 matches would have to be made before the rule is triggered.

Consider now the order of conditional elements shown in the program. The first conditional element is (valid month ?number) and so one match is made. Now when CLIPS checks the second conditional element, (name ?month ?number), only 1 match is possible because CLIPS knows what ?number is. Contrast this with the case described before in which CLIPS found 12 matches to (name ?month ?number) because it didn't know what ?number was and so all 12 months matched. Finally, since there's 1 match to the third conditional element, (start-fact), there is a total of 1 x 1 x 1 = 1 matches required compared to the 12 matches of the other version.

For more information on efficiency, see Appendix C of the CLIPS Reference Manual.

Problems

1. Write a program that will generate a table of squares and cubes of numbers. The user should be asked the math function such as square or cube, the minimum and maximum numbers, and the step size for incrementing numbers. The program should then print out the table. A sample format for an input of math function square, min = 1, max = 5, and step size = 1 is shown below.

```
Number      Square
1           1
2           4
3           9
4           16
5           25
```

2. Write a program which will test if three points are colinear. That is, whether all three points fall on a straight line. The user should be asked to specify the x and y coordinates of the three points in the form

```
Point 1 x1 ?
Point 2 y1 ?
```

and so forth.

After telling the user whether the points are colinear, the user should be asked if they want to do another.

3. Write a program to compute compound interest on a loan using the infix formula

```
Interest=Principal*(1+Interest-rate/Periods/100)**(Periods*Years)
```

where

Principal	:	amount invested
Periods	:	number of times a year that the money is compounded
Interest-rate	:	yearly percent rate

```
   Run the program for
      Principal     = 20
      Periods       = 12
      Years         = 366
      Interest rate = 8
```

to see how much money the Dutch would have made if they had invested their money rather than buying Manhatten Island for $20 in 1620.

4. Write a program that tells the day of the week, i.e., Sunday, Monday, and so forth when a user inputs the month, day, and year. The program should also check for invalid entry. For example, months should be in the range 1 - 12. The day of month will depend on the month. So a day of 31 will be valid if the month is 1 but not if the month is 2. The program should also account for leap years.

Chapter 7 Let's Be Logical

In this chapter, you will learn how CLIPS deals with logical relations and comparisons. There are many useful applications of logical operations, especially when you write programs dealing with symbolic information.

Three Into One Will Go

So far all the rules you've seen have an **implicit logical AND** between the conditional elements. That is, a rule will not be triggered unless all of the conditional elements are true. However, CLIPS also gives you the capability of specifying explicitly **logical AND** conditions and also **logical OR** conditions on the LHS.

As an example of OR, let's first look at the following rules which are written without an OR and then see how to rewrite them with an OR.

```
(defrule dont-walk
   (light red)
=>
   (printout "Don't walk" crlf))

(defrule dont-walk
   (walk-sign dont walk)
=>
   (printout "Don't walk" crlf))

(defrule dont-walk
   (police say dont walk)
=>
   (printout "Don't walk" crlf))
```

Rather than writing three separate rules, they can all be combined into the following OR rule.

```
(defrule dont-walk
   (or
      (light red)
      (walk-sign dont walk)
      (police say dont walk))
=>
   (printout "Don't walk" crlf))
```

The group of conditional elements enclosed within the OR element is called a **logic block.** The one OR rule is equivalent to the three rules above. Enter the OR rule and assert the three facts. If you check the agenda, you'll see that the rule is triggered three times with each of the facts.

Other conditional elements can be included outside the OR conditional and will be part of the implicit AND. For example, if the OR rule was

```
(defrule dont-walk
   (status walking)
   (or
      (light red)
      (walk-sign dont walk)
      (police say dont walk))
=>
   (printout "Don't walk" crlf))
```

the result would be equivalent to the three rules following.

```
(defrule dont-walk1
   (status walking)
   (light red)
=>
   (printout "Don't walk" crlf))

(defrule dont-walk2
   (status walking)
   (walk-sign dont walk)
=>
   (printout "Don't walk" crlf))

(defrule dont-walk3
   (status walking)
   (police say dont walk)
=>
   (printout "Don't walk" crlf))
```

As you can see, the OR is similar to a true logical OR. In a true logical OR, the rule dont-walk would only be triggered once if one or more conditions are true. However, in the OR of CLIPS, the dont-walk rule will be triggered for each OR conditional element that is true, since the one OR rule is equivalent to three rules. However, an OR rule does function like a true logical OR in the sense that if any conditional element matches a fact, then the rule will be triggered.

Since multiple triggerings of an OR will occur with multiple facts, a natural question is how to prevent the problem of more than one triggering. For example, we don't want multiple printouts of "Don't walk" appearing for multiple facts. After all, the robot only deals with one traffic light at a time. Just because multiple facts may be true about the traffic light doesn't mean we want a printout of "Don't walk" for each fact.

One possible solution to multiple triggerings you might think of is to retract all the triggering facts when the rule fires once. If all of the other triggering facts of the OR are retracted, the rule won't be triggered again. For example,

```
(defrule dont-walk
               (or
                  ?fact <- (light red)
                  ?fact <- (walk-sign dont walk)
                  ?fact <- (police say dont walk))
            =>
               (retract ?fact)
               (printout "Don't walk" crlf))
```

Notice that all conditional elements are bound to the same variable, ?fact. At first, you may think this an error since you'd never assign the same variable to conditional elements of an ordinary rule. But an OR rule is different. Since an OR is equivalent to multiple rules, the above rule is equivalent to the following three rules.

```
(defrule dont-walk1
   (status walking)
   ?fact <- (light red)
=>
   (retract ?fact)
   (printout "Don't walk" crlf))

(defrule dont-walk2
   (status walking)
   ?fact <- (walk-sign dont walk)
=>
   (retract ?fact)
   (printout "Don't walk" crlf))

(defrule dont-walk3
   (status walking)
   ?fact <- (police say dont walk)
=>
   (retract ?fact)
   (printout "Don't walk" crlf))
```

By looking at the three rules, you can see that the same variable name was necessary to match the (retract ?fact) action. For example, suppose you had assigned separate names to the conditional elements such as ?light, ?walk, and ?police and then used a (retract ?light ?walk ?police).

```
(defrule dont-walk
   (status walking)
   (or
      ?light <- (light red)
      ?walk <- (walk-sign dont walk)
      ?police <- (police say dont walk))
=>
   (retract ?light ?walk ?police)
   (printout "Don't walk" crlf))
```

The (retract ?light ?walk ?police) would be in error because two out of three variables would not be defined in the equivalent rules.

Even though our rule does retract multiple facts, it still prints out multiple "Don't walk" messages because the one rule is exactly the same as three rules. Try it and see.

The correct solution to prevent multiple messages is to define a control fact. Consider the following program.

```
(defrule dont-walk
   ?control <- (control-fact)
   (status walking)
   (or
      ?fact <- (light red)
      ?fact <- (walk-sign dont walk)
      ?fact <- (police say dont walk))
=>
   (retract ?control ?fact)
   (printout "Don't walk" crlf))
```

If the control fact is retracted, the rule cannot fire again. It's as if you retracted the (status walking) fact. Strictly speaking, we don't even need to retract the ?fact. However, it's a good idea to retract all facts that characterize the current light situation so that the facts don't hang around and accidentally cause a firing if the control fact is re-asserted. That is, what if the robot wanders on and encounters another light. You don't want the old light's facts in memory.

And What's More

The logical AND is opposite in concept to the logical OR. Instead of any conditional elements triggering a rule, the AND requires that all of the conditional elements be matched to facts.

Normally there is an implicit AND for the conditional elements of a rule. A rule such as

```
(defrule and-test
   (status walking)
   (walk-sign walk)
=>
   (printout "Walk" crlf))
```

could also be written with an explicit AND as

```
(defrule and-test
      (and
         (status walking)
         (walk-sign walk))
=>
   (printout "Walk" crlf))
```

where the indentations are just to aid readability.

Of course, there is no advantage to writing a rule with an explicit AND. It's more efficient to use the implicit AND of conditional elements if you don't need to specify an AND.

The AND is provided so that you can use it with the other logical functions to make more powerful patterns. For example, it can be used with an OR to require groups of multiple conditions be true, as shown in the following example.

```
(defrule walk
   (or (police says walk)
       (and (walk-sign walk)
            (light green))
       (and (walk-sign walk)
            (light yellow)))
=>
   (printout "Walk" crlf))
```

Enter this rule and assert the facts

```
(police says walk)
(walk-sign walk)
(light green)
(light yellow)
```

When you check facts, you'll see

```
f-0     (initial-fact)
f-1     (police says walk)
f-2     (walk-sign walk)
f-3     (light green)
f-4     (light yellow)
```

Now check the agenda. The rule will be triggered with the facts as shown following.

```
0   walk   f-2  f-4
0   walk   f-2  f-3
0   walk   f-1
```

So the one rule called walk is equivalent to the three following rules, where the comments identify the triggering facts.

```
(defrule walk
   (walk-sign walk)       ; f-2 and f-4
   (light yellow)
=>
```

```
   (printout "Walk" crlf))

(defrule walk
   (walk-sign walk)        ; f-2 and f-3
   (light green))
=>
   (printout "Walk" crlf))

(defrule walk
   (police says walk)      ; f-1
=>
   (printout "Walk" crlf))
```

Not My Constraint

The previous rule are simplified ones that don't cover all cases such as the breakdown of the traffic-light. For example, what does the robot do if the light is red or yellow and the walk-sign says walk?

One way of handling this case is to use a **field constraint** to restrict the values a pattern may have on the LHS. The field constraints put limits on the values a pattern may have on the LHS.

There are two types of constraints. The first type is called a **logical constraint.** There are three types of logical constraints. The first type of logical constraint is called a **negation constraint.** It's symbol is the "~", called the **tilde.** The negation is a prefix constraint acts on the one value that immediately follows it.

As a simple example of negation, suppose you wanted to write a very conservative rule which would print out "Don't walk" if the light was not green. This constraint is not the same as saying there is no fact (light green) and so a (not) is not appropriate.

One approach would be to write rules for every possible light condition, such as yellow, red, blinking yellow, binking red and so forth. However, a much easier approach is to use the negation constraint, "~", as shown in the following rule.

```
(defrule walk
   (light ~green)
=>
   (printout "Don't walk" crlf))
```

By using the negation constraint, this one rule does the work of many others that required specifying each light condition.
Note that you cannot use the "!" and "!=" functions or any other RHS functions on LHS patterns. Likewise, the logical constraints cannot be used on the RHS.

Be Cautious

The second logical constraint is the **OR constraint.** The OR constraint is represented by the symbol "|", called the **bar**, and is an infix constraint. The OR constraint is used to allow any value in a pattern to match.
For example, suppose you wanted a rule which printed out "Be cautious" if the light was yellow or blinking yellow. Here's how it's done using the "|" constraint.

```
(defrule cautious
   (light yellow|blinking-yellow)
=>
   (printout "Be cautious" crlf))
```

Enter this rule and then assert (light yellow) and (light blinking-yellow). Check the agenda and you'll see that the rule is on the agenda for each fact. Also, notice that the field blinking-yellow is connected by a dash. Since the logical constraints operate on single fields, you can't have (light yellow|blinking yellow) because then the constraint would only be on yellow and blinking.

YAA

The third type of logical constraint is yet another and, the **AND constraint.** Its symbol is the "&", called the **ampersand,** and is an infix constraint. The "&" constraint is normally used only with the other constraints since otherwise it's not of much practical use.
To understand where it's useful, let's first look at an example in which the "&" is not useful.

```
(defrule green-light
   (light green&red)
=>
   (printout "Go" crlf))
```

This rule will never be triggered since there is no possible fact whose light is both green and red. The conditional element will not match on green, red, or even greenred.

Here's another example of where the "&" is useless.

```
(defrule green-light
   (light green&~red)
=>
   (printout "Go" crlf))
```

The conditional element will match a light whose color is green and not red. In this case, the rule can be triggered by the fact (light green) since green is green and not red. However, it's much more efficient to just write the rule as

```
(defrule green-light
   (light green)
=>
   (printout "Go" crlf))
```

since a light that is green cannot be red anyway.

Now let's see an example in which the "&" is useful. Suppose you want to have a rule which will be triggered by a fact which is yellow or blinking yellow. That's easy enough. Just use the OR constraint, as you saw in a previous example.

```
(defrule cautious
   (light yellow|blinking-yellow)
=>
   (printout "Be cautious" crlf))
```

Suppose also that you want to identify the light color.

The solution is to bind a variable to the color that is matched using the "&" and then print out the variable. This is where the "&" is useful, as shown below.

```
(defrule cautious
   (light ?color&yellow|blinking-yellow)
=>
```

```
    (printout "Be cautious" crlf))
```

The variable ?color will be bound to whatever color is matched by the field yellow|blinking-yellow.

The reason for not using a (bind) is because it is a RHS function and so cannot be used on the LHS. Likewise, you cannot use the logical constraints on the RHS.

Problem

Write a program to simulate getting a car to run using the following information. The car has the following major components.

engine
spark plugs
battery
gas tank
wheels

For the car to move requires

gas in the gas tank
oil in the engine
coolant in the engine
electricity in the spark plugs
inflated tires on the wheels

If the gas, oil, coolant, and tires are not present, you must walk to a service station and buy them.

If there is no electricity in the spark plugs, you must check the battery.

If the battery is good, you must call a service station to tow the car to the service station.

If a tire is not inflated, you must check to see if there is a leak.

All four tires must be checked to see if they are inflated before the car can operate.

If a tire has a leak, you must check the trunk to see if there is a jack and a spare tire.

If there is no jack but there is a spare tire, you can borrow a jack from your neighbor.

If there is a jack but no spare tire, you will borrow a tire from your spouse's car, then drive to a service station and buy a spare.

If your car has been towed to a service station and you do not have towing insurance, you must pay $49 from cash, check, or charge sources.

If you are at the service station and do not have a single source of payment to meet the cost of an item or towing, you must pay by combining sources.

For expert advice on how to pay, write a advice rule which will let you pay by combining sources from cash, check, and charge. The following facts should be considered by the rule and then it should tell you what to do. Do not encode the numbers in the following facts in the rule. The rule should bind data from the facts to appropriate variables in the rule.

You must always pay the towing charge in full and should buy as much gas and oil as you can pay for. You must have 1 container of coolant and at least 1 gallon of gas and 1 quart of oil for the car to run. You should try to buy as much as you can without overfilling.

<u>Facts</u>

You do not have towing insurance.

The service station will not accept checks over $49

You have $20 in cash

You do not have a credit card

You have $80 in your checking account

The towing charge is $50

Gas is $1.09 a gallon

Oil is 1.56 a quart

Capacity of gas tank is 15 gallons

Capacity of engine for oil is 5 quarts

A new battery is $42.95

A new tire is $69.96

A container of coolant is $2.98

There is 0.5 gallons of gas

There is no oil

Two of your tires have leaks

You have a spare tire and no jack

You need 1 container of coolant

The battery is bad

You must not overfill the gas tank, engine oil, or coolant

Make a (deffacts) of the appropriate facts and run. The program should tell you what to check and what to do for all the problems that occur. Depending on what you input when you're asked to check something, the program will take appropriate action.

Chapter 8 Powerful Patterns

In this chapter, you will learn more of the powerful pattern matching capabilities of CLIPS. By effectively using this capability you can greatly reduce the number of rules required by programs and also make them more easily understandable.

Stronger Than Logic

In the previous chapter, you learned about one type of field constraint, the logical constraint which uses "&", "|", or "~". The second type of field constraint is called a **predicate function,** and is commonly used for more complex pattern matching of a field. You'll find the predicate function very useful with numeric patterns.

A predicate function uses a special form of the "&" constraint called the **AND function** and symbolized by "&:". The general form is

```
?variable&:(<fun> <<arg>>)
```

where
?variable is any legal variable name
<fun> is a predefined or user defined function
<<arg>> are zero or more arguments required by the function

Any predefined or user defined function can be used as a function. The predefined functions are those shown in the section on the test function in the section "Control Your Loop" of Chapter 6, and also in Appendix A of the Reference Manual. The math functions can also be used if you have a version of CLIPS which has been linked to the math library.

The "&:" function operates by initially binding the value in the field to ?variable and then performing the test (<fun> <<arg>>). If the test is succesful, the conditional element is true.

As a very simple example, consider the following.

```
(defrule compare
   (?x&:(> 2 1))
   =>
   (printout "answer " ?x crlf))
```

Enter and assert a 1. When you run, the same fact will be printed. In fact, the rule will always print out whatever facts are asserted because the predicate function test is always true. That is, since 2 > 1 is always true, any fact meets the criteria of the predicate function and will be bound to ?x.

A more useful form of the predicate function will use the variable as one of the arguments. For example, suppose you wanted to test which numbers are greater than 1. The modified rule would now be

```
(defrule compare
   (?x&:(> ?x 1))
   =>
   (printout "answer " ?x crlf))
```

Before you enter this rule, do not have an (initial-facts) or other non-number in the fact-list. The reason is that CLIPS will try to match the conditional element on (initial-fact). So give an error message because (initial-fact) is not a number and the test (> ?x 1) is invalid. If there is a non-numeric fact such as (initial-fact), you'll see an error message

```
Non-float argument #1 found in function > in rfloat call
```

With no non-numeric fact present, enter the rule and assert the numbers 0, 1, 2, 3, and 4. When you run, you'll see that the numbers greater than 1 are indeed printed out.

In order to prevent error messages like the above, you can either try never to have a non-numeric fact or tighten up the constraints on the facts to be matched. For example, the rule could be changed to

```
(defrule compare
   (number ?x&:(> ?x 1))
   =>
   (printout "answer " ?x crlf))
```

and then you would assert facts like (number 0), (number 1), and so forth. Now (initial-fact) cannot match the conditional element and the error message does not occur.

Notice that the "&:" constraint now applies to the second field in the conditional element. Using constraints, wildcards, and tests gives you a very powerful pattern matching capability.
The presence of the extra "number" pattern in the first field means that CLIPS must do a little more matching to trigger a rule. However, the extra matching time is insignificant. The important thing is that extra fields aid in debugging a program because they explicitly show what the facts are supposed to be.
You may be wondering why the second version of the program involving ?x produced the error message and the first with (> 2 1) did not. The reason is that CLIPS did not have to test ?x in (> 2 1) and so did not find an error in comparing initial-fact to 1.
Now suppose you want those numbers which are greater than 1 and less than 4. No sweat. Just add the following conditional element to check if ?x is < 4.

```
(defrule compare
   (number ?x&:(> ?x 1))
   (number ?x&:(< ?x 4))    ; add this conditional element
   =>
   (printout "answer " ?x crlf))
```

After you change the rule, check the agenda. Notice that the facts for the numbers 2 and 3 are listed twice since they match the two conditional elements of the rule. If nothing is on the agenda, you probably didn't assert the facts with a "number" in the first field. When you run, the rule will print out the correct numbers 2 and 3.
Is there any way to write the rule using only one conditional element? There is, as shown by the following rule.

```
(defrule compare
   (number ?x&:(&& (> ?x 1)) (< ?x 4))
   =>
   (printout "answer " ?x crlf))
```

The one conditional element says that ?x must be greater than 1 and less than 4. Notice that the logical AND, "&&", must be used as the connector for the two comparisons (> ?x 1) and (< ?x 4).

The Truth Of The Matter

A function that returns a value of 0 is considered false, while non-zero is true. As an simple example, consider the following.

```
(defrule and-1
   (number ?x&:1)
   =>
   (printout "true" crlf))

(defrule and-0
   (number ?x&:0)
   =>
   (printout "false" crlf))
```

Notice that there is a number after the "&:" rather than a function and arguments in parentheses. The number alone indicates the value that would have been returned by a function. An alternative is a function that is always true like (= 1 1) or one that is always false like (= 1 0).

Enter these rules and then do a (reset). As expected, the and-1 is on the agenda while the and-0 is not. If you replace the 1 and 0 after the "&:" with the always true and always false function, you'll see the same results.

How would you invert the logic of the function result to make a false become true? The answer is shown in the following modification of the and-0 rule. The logical not, "!", is used to invert the 0. Enter this rule and (reset). You'll see that it is on the agenda.

```
(defrule and-0-true
   (number ?x&:(! 0))
   =>
   (printout "false" crlf))
```

The same considerations of true and false apply to test conditions. For example, enter the following, (reset), and check the agenda.

```
(defrule test-1
   (number ?x)
   (test 1)
   =>
   (printout "true" crlf))
```

```
(defrule test-0
   (number ?x)
   (test 0)
   =>
   (printout "false" crlf))
```

Normally you'd never intentionally use constant functions like this since they are not of practical use. However, you might accidentally write one involving variables such as

```
(defrule error
   (number ?x&:(> ?x ?x))
=>
   (printout "false" crlf))
```

The programmer probably meant to write a conditional element like (number ?x&:(> ?x ?y)) but accidentally typed a ?x. Since a number cannot be greater than itself, the function is always false and the rule is never triggered.

If you write your own functions, they must return 0 for false and a nonzero value for true. Any nonzero number will do.

You Just Can't Get Enough

Besides the predefined functions, there are some other **predicate functions** that are very useful, especially with "&:". These predicate functions are used to test strings and numbers. As the old saying goes, you just can't get enough of a good thing. Notice that there is a "p" after each function. The reason for the "p" is that these functions were originally defined in LISP and that's how LISP defines them.

Function	Purpose
(numberp <arg>)	**Check if <arg> is a number**
(stringp <arg>)	**Check if <arg> is a string**
(evenp <arg>)	**Check if <arg> is even**
(oddp <arg>)	**Check if <arg> is odd**

As an example, consider the following rules to print out the odd and even numbers in the fact-list. Note that there may also be non-numbers, but only the numbers will be printed out.

```
(defrule odd-number
   (?x&:(&& (numberp ?x) (oddp ?x)))
=>
   (printout ?x " is an odd number" crlf))

(defrule even-number
   (?x&:(&& (numberp ?x) (evenp ?x)))
   =>
   (printout ?x " is an even number" crlf))
```

The "number" field is not necessary now because the "numberp" checks to see if the fact is a number.

Enter and assert the facts duck, cat, -2, -1, 0, 1, and 2. Check the agenda and you'll see that the rules do match the correct numbers.

Pushing CLIPS

Let's take a look at another example. This one will really push CLIPS by straining its pattern matching and arithmetic capability.

Suppose you want a program that will determine if three numbers are perfect squares. That is, what integer numbers x, y, and z will satisfy the equality

```
z * z = x * x + y * y
```

First of all, we'll have to agree on some limits to the problem. Since there are infinitely many numbers that satisfy the equality, we'll let the user pick a certain range of numbers for CLIPS to test.

The program will be written as three rules. The input-limits rule will let the user specify the initial and final values of numbers. The make-numbers rule will then assert the integers from initial to final value. The perfect-squares rule will then find the perfect squares from the numbers asserted by the make-numbers rule. Shown following are the first two rules of the program.

```
(defrule input-limits
   (initial-fact)
=>
   (printout "Initial number ? " crlf)
   (bind ?initial (read))
   (printout "Final number ? " crlf)
   (assert (limits =(read) ?initial))) ;final and initial limits

(defrule make-numbers
   (declare (salience 10))
   ?limits <- (limits ?final ?initial&:(<= ?initial ?final))
=>
   (retract ?limits)
   (printout "number " ?initial " asserted" crlf)
   (assert (number ?initial))
   (bind ?initial (+ ?initial 1))
   (assert (limits ?final ?initial)))
```

Enter these two rules and run for an initial limit of 1 and a final limit of 7. Notice how quickly CLIPS prints the asserted numbers. Now enter a (reset) command. Then enter the third rule to find numbers which are perfect squares.

```
(defrule perfect-squares
   (number ?x)
   (number ?y)
   (number ?z&:(= (* ?z ?z) (+ (* ?x ?x) (* ?y ?y))))
           =>
   (printout "x = " ?x " y = " ?y " z = " ?z crlf))
```

The salience conditional element in the make-numbers rule is to insure that all the asserted numbers are asserted before the perfect-squares rule is triggered. Now run the program for the same limits as before of 1 and 7.

The third conditional element in perfect squares is the equality to be tested. It checks if the sum of the square of z is the sum of the squares of x and y. Enter and run the program as before for initial and final numbers of 1 and 7. Notice that it takes CLIPS longer and longer to assert the numbers now that the perfect-squares rule is present.

The reason for the delays in asserting numbers is that CLIPS also checks which rules are triggered every time a new fact is asserted. Since there are more facts, there are more comparisons to check. Try to calculate how many checks of facts against conditional elements there are as each new fact is added.

As soon as a fact is asserted, CLIPS starts checking for matches against rules. In other languages such as FORTAN, Pascal, Ada, and C, nothing happens until you issue a run command. But in CLIPS and LISP, the interpreter is always running. The situation is analogous to BASIC in which the interpreter is always running and will immediately execute a command without a line number. In CLIPS, the interpreter automatically matches rules against facts, even before your program starts executing.

Now you can see why you were asked to enter and run the first two rules before the third rule. This way, you have gained a better understanding of how CLIPS operates than if you had just typed in all three rules at once.

A more efficient method of writing the program is to eliminate the salience conditional element in the make-numbers rule. The salience was included to just illustrate the difference in execution speed before and after the perfect-squares rule was added. Try running the program without salience also.

Problems

1. Write a program that will first assert the numbers from 1 to 24 and then determine which are factorials.

2. Write a program to give directions to a robot on how to go from point 1 to point 2 by the shortest path. The directions will tell how to move one square at a time in the form

```
move north
move south
move east
move west
```

Different traffic lights and buildings will be asserted from a (deffacts), whose facts are in the form

```
(building x y)

(light red x y)

(light green x y)
```

and so forth for the other light conditions. The x and y coordinates are shown followed by a diagram. Use the x and y coordinates for the buildings and lights shown. The robot starts at x = 0, y = 0. The goal is at x = 9, y = 9. Ideally, the robot will travel in a straight line to the goal.

For each move, print out the direction to travel and update the diagram. The symbols on the diagram represent the following.

B	building
O	robot
X	goal
r	red light
y	yellow light
g	green
G	blinking green - The light is broke. Cannot go in this square
Y	blinking yellow
R	blinking red

The robot must be in a square to know what is in the immediately surrounding squares. That is, the robot can only look at the immediately adjacent eight squares. It cannot go past the borders, into a building, or into a broken light.

Hint: If you can't go in a straight line, then always go in one consistent direction, such as right or left.

```
                          ------------
                         9|      BB X|
                         8|   GB  r  |
          N              7|   BBBB   |
        W   E            6|  R  YBB  |
          S           Y  5|  g BrBBB |
                         4|  B  y  B |
                         3|BBB  B   B|
                         2|  BBBg BBB|
                         1|   Br B   |
                         0|O         |
                          ------------
                           0123456789

                               X
```

Chapter 9 Backwards And Forwards

Up to this point, you've been learning the basic syntax of CLIPS. In this chapter you will learn more advanced techniques of overall program design.

Forward And Backward Chaining

There are two fundamental **control strategies** that a program can follow. One type is called **forward chaining** and the other is called **backward chaining.** The control strategies should not be confused with controlling a loop. Instead, the control strategy of a program refers to the most basic design philosophy of the program.

The influence of these control strategies is so important that languages are usually designed to optimize one or the other. For example, CLIPS is designed as a forward chaining language while PROLOG is backward chaining. With appropriate programming, a forward chaining language can be used for backward chaining and vice versa. So far the programs you've been writing have been forward chaining.

Forward chaining reasons from known facts to the resulting conclusions. In contrast, backward chaining reasons from known conclusions to the facts which caused them. The determination of which control strategy to use depends on the application.

Consider the problem of writing an expert system program to diagnose illness. If a patient has certain symptoms, the program should try to determine the nature of the illness which caused the illness. In this case, a backward chaining strategy is desired. In contrast, if a patient is known to have a certain disease, an expert system program could be written for a prognosis of the disease. That is, the program could predict the course of the disease and suggest a therapy.

Another consideration in the choice of forward or backward chaining strategies is the extent of the **search space.** The search space is the set of possible states a program must search to find an acceptable solution. The term **state** means the collection of information which defines a configuration of the system.

For many problems, a brute-force approach by searching all possible states is impractical because there are too many states. Instead, the production rules contain knowledge about which paths to follow in the search space that are likely to lead to the desired goal.

Sometimes it is necessary for the program to **backtrack** to a state that was searched earlier to take an alternate path from that state because the first path chosen did not lead to an acceptable state. An example is searching through a maze and coming to a dead end. You would then go back along your path and take a different path.

The knowledge encoded in rules may be **heuristic** in nature. The term heuristic means a method that may aid in the solution of a problem. A heuristic is not guaranteed to aid in the solution. Only an algorithm is guaranteed to determine a solution. Very often the knowledge of experts is heuristic. The person writing an expert system program, the **knowledge engineer,** must interview the expert and implement this knowledge in an expert system program.

In a good forward chaining application, there are many possible final states and few initial states. Information is known about the initial states and the problem is to find an acceptable path from the initial state to the final state. Forward chaining reasons from given facts to the conclusions which follow from them. The production rules in CLIPS are written in a forward chaining syntax. The known facts are the conditional elements which imply the conclusions which are the actions of the rule.

In a good backward chaining case, there are relatively few final states and many initial states. The goal in backward chaining is to reason backward from conclusions to the facts which caused them.

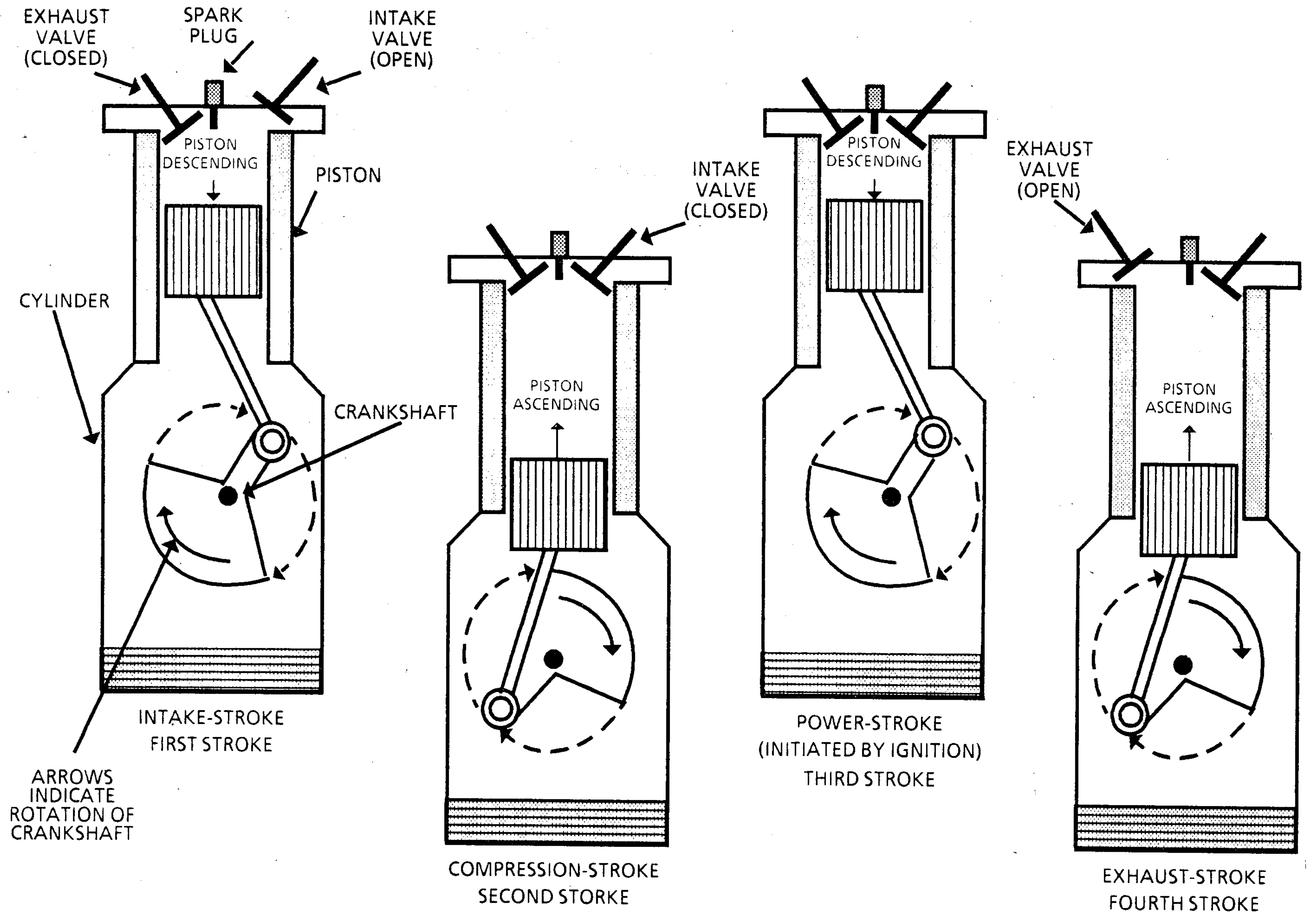
A Forward Engine

Let's first look at an example of an application written in the standard forward chaining method of CLIPS and then see it written in a backward chaining format.

The following program simulates the action of a one-cylinder, four-stroke gasoline engine. Figure 9-1 illustrates the basic operation of the engine. The engine works in the following way.

FIGURE 9-1

FOUR STROKES OF A ONE-CYLINDER ENGINE



C-2

1st Stroke-Intake Stroke: While the intake valve is open and the exhaust valve is closed, the descending piston draws a fresh air and gasoline fuel mixture into the cylinder.

2nd Stroke-Compression Stroke: The intake valve closes and the ascending piston compresses the fuel mixture.

Ignition: The compressed mixture is ignited by an electrical spark from the spark plug.

3rd Stroke-Power Stroke: The pressure of the burned gases from the mixture forces the piston to descend.

4th Stroke-Exhaust Stroke: The exhaust valve is opened and the ascending piston expels the burned gases from the cylinder.

As the piston ascends and descends, it rotates the crankshaft which supplies mechanical power to the wheels. The crankshaft is a rod going into the paper and symbolized by the black dot.
Enter the following program which simulates the operation of the engine.

```
;***** forward engine program *****
;forward chaining model of a one-cylinder, four-stroke
;gasoline engine

(defrule make-initial-fact
   (initial-fact)
=>
   (assert (engine intake-valve open
                   exhaust-valve closed
                   gas good
                   air good
                   sparkplug nofire
                   piston descending)))

(defrule intake-stroke
   ?engine <- (engine intake-valve open
                      exhaust-valve closed
                      gas good
                      air good
                      sparkplug nofire
                      piston descending)
=>
```

```
   (printout crlf "1st stroke" crlf)
   (retract ?engine)
   (assert (engine intake-valve closed
                   exhaust-valve closed
                   gas good
                   air good
                   sparkplug nofire
                   piston ascending)))

(defrule compression-stroke
   ?engine <- (engine intake-valve closed
                      exhaust-valve closed
                      gas good
                      air good
                      sparkplug nofire
                      piston ascending)
=>
   (printout "2nd stroke" crlf)
   (retract ?engine)
   (assert (engine intake-valve closed
                   exhaust-valve closed
                   gas good
                   air good
                   sparkplug fire
                   piston ascending)))

(defrule ignition
   ?engine <- (engine intake-valve closed
                      exhaust-valve closed
                      gas good
                      air good
                      sparkplug fire
                      piston ascending)
=>
  (printout "ignition" crlf)
  (retract ?engine)
  (assert (engine intake-valve closed
                  exhaust-valve closed
                  gas burned
                  air burned
                  sparkplug nofire
                  piston descending)))

(defrule power-stroke
   ?engine <- (engine intake-valve closed
                      exhaust-valve closed
                      gas burned
                      air burned
                      sparkplug nofire
                      piston descending)
=>
   (printout "3rd stroke" crlf)
   (retract ?engine)
   (assert (engine intake-valve closed
                   exhaust-valve open
                   gas burned
                   air burned
                   sparkplug nofire
                   piston ascending)))

(defrule exhaust-stroke
   ?engine <- (engine intake-valve closed
                      exhaust-valve open
                      gas burned
                      air burned
                      sparkplug nofire
                      piston ascending)
=>
  (printout "4th stroke" crlf)
  (retract ?engine)
  (assert (engine intake-valve open
                  exhaust-valve closed
                  gas good
                  air good
                  sparkplug nofire
                  piston descending)))
```

To run the engine, (reset) and issue a (run 21). The partial output of the program should look like the following for one cycle and the start of another.

```
1st stroke
2nd stroke
ignition
3rd stroke
4th stroke

1st stroke
```

2nd stroke

A Backward Example

The basic algorithm of the backward chaining method can be stated recursively as follows.

```
Solve the goal: If the goal has been solved
                 then return
                 else make an appropriate subgoal
                      Solve the goal
```

In backward chaining, the program tries to satisfy a **goal**. If the goal cannot be satisfied directly, the program creates one or more new goals, called **subgoals**. The goal is said to be split up into simpler subgoals for solution. The goal which is split is called the **parent goal** and the subgoals are also called **child goals**. The subgoals are designed so that their solution should enable the parent goal to be satisfied.

A large complex problem may have many subgoals created. Of course, not all the subgoals will be active at once. As appropriate subgoals are satisfied, the parent goal is satisfied. When appropriate parent goals are satisfied, their parent is satisfied. If a subgoal cannot be solved or leads to an undesirable consequence, that information must be relayed back to the parent goal so that some other action by the parent is taken.

For the example of the four-stroke engine, the initial goal is to produce an engine cycle. That is, the first goal is to have the engine complete a complete cycle of four strokes. To complete a cycle means that the exhaust-stroke must be done. To complete the exhaust-stroke means that a power-stroke must be done. To complete a power-stroke means that the ignition must be done. To complete ignition requires that a compression-stroke must be done. To complete a compression-stroke requires that an intake-stroke must be done.

As you can see by this analysis, the backward chaining method is a very appropriate name. Each goal generates a subgoal in a backward manner from the desired initial goal of an engine cycle to generating an intake goal. The goals act as if they linked by a chain. In fact, the chain can be considered as the chain of logical inferences which specified the creation of subgoals to satisfy the parent goal.

When each subgoal is created, the program checks to see if it has enough information to solve the subgoal. In the case of our engine, subgoals are created until there is a subgoal for an intake stroke. The initial conditions of the engine are such that it is set up for an intake stroke. That is, the make-initial-fact rule will set the intake-valve to open, the exhaust-valve to closed, the air good, the gas good, the sparkplug to nofire and the piston to descending. The intake-stroke rule will be designed to fire if there is a goal whose parent is split-compression-stroke and whose name is intake-stroke and there is a engine element with the above initial configuration.

The following program shows a backward chaining version of the engine. The four rules which split goals operate first. Then the engine fires the four strokes. Finally, the continue-cycle rule initial-facts another engine cycle. As before, you'll need to issue a (run 21) command to stop after 21 cycles. Enter and run this program. You should also issue (watch facts) and (watch activations) commands to watch facts and activations.

```
;***** backward engine program *****
;backward chaining model of a one-cylinder, four-stroke
;gasoline engine

(defrule make-initial-fact
   (initial-fact)
=>
   (assert (engine intake-valve open
                   exhaust-valve closed
                   gas good
                   air good
                   sparkplug nofire
                   piston descending))
   (assert (goal parent complete-cycle
                 name exhaust-stroke
                 action nil)))

(defrule split-exhaust-stroke
   (goal parent complete-cycle
         name exhaust-stroke
         action nil)
=>
   (assert (goal parent split-exhaust-stroke
                 name power-stroke
                 action nil)))
```

```
(defrule split-power-stroke
   (goal parent split-exhaust-stroke
         name power-stroke
         action nil)
=>
   (assert (goal parent split-power-stroke
                 name ignition
                 action nil)))

(defrule split-ignition
   (goal parent split-power-stroke
         name ignition
         action nil)
=>
   (assert (goal parent split-ignition
                 name compression-stroke
                 action nil)))

(defrule split-compression-stroke
   (goal parent split-ignition
         name compression-stroke
         action nil)
=>
   (assert (goal parent split-compression-stroke
                 name intake-stroke
                 action nil)))

(defrule intake-stroke
   ?goal <- (goal parent split-compression-stroke
                  name intake-stroke
                  action nil)
   ?engine <- (engine intake-valve open
                      exhaust-valve closed
                      gas good
                      air good
                      sparkplug nofire
                      piston descending)
=>
   (printout crlf "1st stroke" crlf)
   (retract ?engine)
   (assert (engine intake-valve closed
                   exhaust-valve closed
                   gas good
                   air good
```

```
                    sparkplug nofire
                    piston ascending))
   (retract ?goal)
   (assert (goal parent split-compression-stroke
                 name intake-stroke
                 action delete)))

(defrule compression-stroke
   ?goal <- (goal parent split-ignition
                  name compression-stroke
                  action nil)
   ?engine <- (engine intake-valve closed
                      exhaust-valve closed
                      gas good
                      air good
                      sparkplug nofire
                      piston ascending)
=>
   (printout "2nd stroke" crlf)
   (retract ?engine)
   (assert (engine intake-valve closed
                   exhaust-valve closed
                   gas good
                   air good
                   sparkplug fire
                   piston ascending))
   (retract ?goal)
   (assert (goal parent split-ignition
                 name compression-stroke
                 action delete)))

(defrule ignition
   ?goal <- (goal parent split-power-stroke
                  name ignition
                  action nil)
   ?engine <- (engine intake-valve closed
                      exhaust-valve closed
                      gas good
                      air good
                      sparkplug fire
                      piston ascending)
=>
   (printout "ignition" crlf)
   (retract ?engine)
   (assert (engine intake-valve closed
                   exhaust-valve closed
                   gas burned
                   air burned
                   sparkplug nofire
                   piston descending))
   (retract ?goal)
   (assert (goal parent split-power-stroke
                 name ignition
                 action delete)))

(defrule power-stroke
   ?goal <- (goal parent split-exhaust-stroke
                  name power-stroke
                  action nil)
   ?engine <- (engine intake-valve closed
                      exhaust-valve closed
                      gas burned
                      air burned
                      sparkplug nofire
                      piston descending)
=>
   (printout "3rd stroke" crlf)
   (retract ?engine)
   (assert (engine intake-valve closed
                   exhaust-valve open
                   gas burned
                   air burned
                   sparkplug nofire
                   piston ascending))
   (retract ?goal)
   (assert (goal parent split-exhaust-stroke
                 name power-stroke
                 action delete)))

(defrule exhaust-stroke
   ?goal <- (goal parent complete-cycle
                  name exhaust-stroke
                  action nil)
   ?engine <- (engine intake-valve closed
                      exhaust-valve open
                      gas burned
                      air burned
                      sparkplug nofire
```

```
                            piston ascending)
=>
   (printout "4th stroke" crlf)
   (retract ?engine)
   (assert (engine intake-valve open
                   exhaust-valve closed
                   gas good
                   air good
                   sparkplug nofire
                   piston descending))
   (retract ?goal)
   (assert (goal parent complete-cycle
                 name exhaust-stroke
                 action delete)))

(defrule continue-cycle
   (engine $?)
   (not (goal $?))
=>
   (printout crlf "***** Engine cycle complete *****")
   (assert (goal parent complete-cycle
                 name exhaust-stroke
                 action nil)))

(defrule remove-satisfied-goal
   ?goal <- (goal $? action delete)
=>
  (retract ?goal))
```

Strictly speaking, not all of the patterns need be written so explicitly for the (engine) in all the strokes. They are written out to just help you understand what's happening in the engine. For example, in the exhaust-stroke, the conditional element for ?engine could be written more correctly with wildcards as

```
   ?engine <- (engine intake-valve closed
                      exhaust-valve open
                      gas ?
                      air ?
                      sparkplug ?
                      piston ascending)
```

because the exhaust-stroke does not depend on the state of the gas, air, or sparkplug. In fact, the conditional element could even be written more briefly as

```
?engine <- (engine intake-valve closed
                   exhaust-valve open
                   $?
                   piston ascending)
```

Specifying the engine patterns more explicitly does help reduce the work that CLIPS does in pattern matching. When wildcards are used, CLIPS does a lot more work because it must try all possible matches. Even with wildcards, the "?" is less work for CLIPS than "$?" and so it's better to use "?".

Give a (run 21) command and observe the speed at which the program executes. You'll see the same kind of output as in the forward chaining version. However, notice that the backward chaining version runs slower. Also, there is less output for a given number of run cycles because CLIPS is doing more work compared to the forward chaining version. These factors slow down the execution speed of the backward version.

A Good Explanation

In this particular example of the four-stroke engine, the backward chaining is not the best choice of a design strategy. Since the strokes of the engine follow each other in a sequential manner, the simulation problem is inherently forward chaining and should best be done by a forward chaining program.

However, the backward chaining method is useful in providing an **explanation facility**. The term explanation facility means that a program can explain its reasoning as it fires rules. That is, the program can explain why it is trying to satisfy certain goals. An explanation facility is a very important feature in both program development and also in real runs by users.

As the number of rules and facts grow, it becomes harder for a programmer to just look at the code and understand how a program works. In contrast to programs written in sequential languages like Pascal, rule-based programs execute in a parallel manner since all the rules have simultaneous access to the working memory. The solution to this problem is to use the program to debug itself. By designing with parent and subgoals in mind, the program can tell its reasons to perform

certain actions.
   Let's add an explanation facility to the Backward Engine Program so that you can see how it works. Modify the make-initial-fact rule as follows so that it will ask the user if an explanation is desired.

```
(defrule make-initial-fact
   (initial-fact)
=>
   (assert (engine intake-valve open
                   exhaust-valve closed
                   gas good
                   air good
                   sparkplug nofire
                   piston descending))
   (printout "Explain goals (yes or no) ?" crlf) ;* add this *
   (assert (explain =(read)))                    ;* add this *
   (assert (goal parent complete-cycle
                 name exhaust-stroke
                 action nil)))
```

   Now add the following rule to explain the creation of subgoals. Notice the salience fact so that this rule will fire before a competing split type rule in the agenda. After you run the program with this salience, try running it with no salience and note the difference in output.

```
(defrule explain-subgoal
   (declare (salience 10))
   (explain yes)
   (goal parent ?parent
         name ?name
         action nil)
=>
   (printout "Must create subgoal :" ?name crlf
             "in order to satisfy parent :" ?parent crlf))
```

   Also, add this rule to explain the satisfaction of subgoals as the stroke rules fire.

```
(defrule explain-remove-satisfied-goal
   (explain yes)
   ?goal <- (goal parent ?parent
                  name ?name
```

```
                    action delete)
=>
   (retract ?goal)
   (printout "Subgoal: " ?name " was satisfied" crlf
             "Now satisfying parent : " ?parent crlf))
```

Finally, modify the remove-satisfied-goal rule by adding the (explain no) condition element as follows.

```
(defrule remove-satisfied-goal
   (explain no)
   ?goal <- (goal $? action delete)
=>
   (retract ?goal))
```

Now do a (reset) and (run 22). Answer "yes" to the explain question as shown and you'll see the following output of one complete engine cycle and start of another.

```
Explain goals (yes or no) ?
yes
Must create subgoal :exhaust-stroke
in order to satisfy parent :complete-cycle
Must create subgoal :power-stroke
in order to satisfy parent :split-exhaust-stroke
Must create subgoal :ignition
in order to satisfy parent :split-power-stroke
Must create subgoal :compression-stroke
in order to satisfy parent :split-ignition
Must create subgoal :intake-stroke
in order to satisfy parent :split-compression-stroke

1st stroke
Subgoal: intake-stroke was satisfied
Now satisfying parent : split-compression-stroke
2nd stroke
Subgoal: compression-stroke was satisfied
Now satisfying parent : split-ignition
ignition
Subgoal: ignition was satisfied
Now satisfying parent : split-power-stroke
3rd stroke
Subgoal: power-stroke was satisfied
Now satisfying parent : split-exhaust-stroke
```

```
4th stroke
Subgoal: exhaust-stroke was satisfied
Now satisfying parent : complete-cycle
***** Engine cycle complete *****

Must create subgoal :exhaust-stroke
in order to satisfy parent :complete-cycle
```

What's The Trouble

In addition to providing an explanation of goals, a program can also be used for **diagnostic tasks** by adding rules that look for problems. For example, suppose the compression-stroke rule is changed to give a bad sparkplug action. In a system which had inputs from engine sensors, the bad sparkplug information could come directly from a sensor and be accepted into the rule. You can simulate the engine sensor data by changing the sparkplug action in the rule as follows to give a bad sparkplug.

```
(defrule compression-stroke
   ?goal<-(goal parent split-ignition
                name compression-stroke
                action nil)
   ?engine<-(engine intake-valve closed
                    exhaust-valve closed
                    gas good
                    air good
                    sparkplug nofire
                    piston ascending)
=>
   (printout "2nd stroke" crlf)
   (retract ?engine)
   (assert (engine intake-valve closed
                   exhaust-valve closed
                   gas good
                   air good
                   sparkplug bad ; changed from fire to bad
                   piston ascending))
   (retract ?goal)
   (assert (goal parent split-ignition
                 name compression-stroke
                 action delete)))
```

Now when the compression-stroke rule fires, it produces a working memory element with a bad sparkplug. To detect a bad sparkplug, let's add a diagnostic rule to look for a bad sparkplug.

```
(defrule bad-sparkplug
   (goal parent ?
         name ignition
         action ?)
   ?engine <- (engine intake-valve closed
                      exhaust-valve closed
                      gas good
                      air good
                      sparkplug bad
                      piston ascending)
=>
   (printout "????? No ignition ?????" crlf
             "Type fire for ignition" crlf)
   (retract ?engine)
   (assert (engine intake-valve closed
                   exhaust-valve closed
                   gas good
                   air good
                   sparkplug =(read)
                   piston ascending)))
```

If the sparkplug is bad, the ignition rule won't fire. But that doesn't tell us why the ignition didn't fire. The rule above, bad-sparkplug, will look for a bad sparkplug and tell us the reason. Additional rules could be added to diagnose other conditions such as bad gas or bad air. As more diagnostic rules are added, the program acts more and more like an expert. In fact, expert system programs are usually designed to grow in an incremental fashion like this.

In a conventional sequential program, we usually have a good idea of the algorithm to be programmed. However, in an expert system, we may not have a good algorithm because our knowledge is incomplete or heuristic. The goal in expert system programming is basically **rapid prototyping**. That is, we want to produce a prototype program quickly. This consideration is very important if the source of knowledge is a human expert. A knowledge engineer may spend a lot of time interviewing the expert and trying to extract the knowledge needed for the program. The expert is the best judge of the performance of the program and it is important to provide rapid feedback to the expert so that the system can be improved.

Problem

Write a program to play tic-tac-toe for these criteria.
(a) The program should never lose. It will either win or draw.
(a) Display the board on the screen with unfilled squares numbered in the following pattern, and ask the user for the number of the square that the user wishes to move to. The user always goes first with an X.

```
   1  |  2  |  3
-----------------
   4  |  5  |  6
-----------------
   7  |  8  |  9
```

(b) After each move, the board should be redrawn and updated.
(c) The program should print out messages for a draw or win.
(d) If the user inputs "why" after the computer moves, the computer should explain why it moved to that square in terms of its goals and which squares it plans to win on.

Print out three games showing computer wins and three draws. In one game of each type, ask a "why" question after moves.

Hint: The board can be considered to have values associated with each square that add up to a magic square of 15.

```
   8  |  3  |  4
-----------------
   1  |  5  |  9
-----------------
   6  |  7  |  2
```

The goal is to move pieces to obtain a sum of 15 so that you can win while preventing your opponent from getting 15.

Optional: (1) The program should give the user a initial-facting choice of X or O and choice of moving first. (2) An error message is printed if the user specifies a number out of the range 1-9 or already taken. (3) The program should be able to predict a draw.

Chapter 10 Adding Structure

In this chapter you'll see alternate ways to write loops and other program structures than the standard method of multiple rules. While these method do work, there is also an inherent danger because your program design can become very inefficient.

If This Goes On

CLIPS provides some programming structures that can be used on the RHS. These structures are the **while** and **if then else** that are also found in modern high-level languages such as Ada and Pascal.

The basic (if then else) has the form

```
(if (<fun> <<arg>>)
 then
   (<<actions>>)
 else
   (<<actions>>)
```

where <fun> stands for a predefined or user-defined function, and <<arg>> and <<actions>> stand for one or more arguments and actions.

If the condition (<fun> <<arg>>) is true, then the actions in the (then) are executed. Else, if the condition is false, the actions in the (else) are executed. Actually, the (else) is optional and so you can also just write an (if then) rule with no (else) part. Once the (if then else) completes execution, CLIPS continues with the next action, if any, on the RHS.

As a simple example of (if then else), enter the following rule. Then assert (light green), then (light red) and run.

```
(defrule traffic-light
   (light ?color)
=>
   (if (eq ?color green)
    then
       (printout "Go" crlf)
    else
```

```
        (printout "light not green" crlf)))
```

If the light is green, the (then) part will be executed and "Go" will be printed out. If the light is not green, the message "light not green" will be printed out.

The (if then else) can be **nested.** That is, one (if then else) can be included within another. The following example shows how one rule can handle all three cases of the traffic light. The indentation of the (if) parts is to aid readability. Enter and run this for the facts (light green), (light red), and (light yellow).

```
(defrule traffic-light
   (light ?color)
=>
   (if (eq ?color green)
    then
       (printout "Go" crlf)
    else
       (if (eq ?color yellow)
        then
           (printout "Slow down" crlf)
        else
           (if (eq ?color red)
            then
               (printout "Stop" crlf)
           )
       )
   )
)
```

Notice that the closing right parentheses are lined up under their corresponding left parentheses. When you have a lot of nesting, it's very helpful to someone reading the program to see the parentheses enclosing the different parts of the rule. It's also easier on the programmer writing the code to tell when all the parentheses are balanced.

Although the above rule does work, it is shown just as an example of (if then else) syntax. The program should really be written as three rules to make efficient use of CLIPS. The danger of using an (if then else) is that you tend to think in a procedural fashion and so write your programs that way. However, CLIPS is not a procedural language. To make the best use of CLIPS, you should think in terms of rules and not try to structure a CLIPS program in a procedural way.

Meanwhile

The second type of structure provided by CLIPS is the (while). Although you can use (test) and multiple rules to produce a loop, the (while) structure can be used to produce a loop within a single rule on the RHS. The general form is

```
(while (<fun> <<arg>>)
         (<<action>>))
```

where <<action>> stands for one or more actions and <<arg>> are the zero or more arguments required by the function, <fun>. The actions of the (while) comprise the **body** of the loop.

The part of the (while) represented by (<fun> <<arg>>) represents a condition that is tested before the actions in the body are executed. If the while-condition is true, then the actions in the body will be executed. If the while condition is false, execution continues with the next statement after the (while), if any.

As an example of a while-loop, let's look at the program to generate squares of numbers written in a while-loop form. Compare the following program with that written in the first section of Chapter 6.

```
(defrule input-max
   (start-fact)
=>
   (printout "number of loops? " crlf)
   (bind ?max (read))
   (bind ?count 0)    ;initialize ?count to 0
   (while (<= ?count ?max)
           (bind ?square (* ?count ?count))
           (printout "square of " ?count " is " ?square crlf)
           (bind ?count (+ ?count 1))))
```

The actions in the while-loop body will continue printing out the squares of numbers until ?count is greater than ?max because then the while-condition, (<= ?count ?max), will be false. Enter and run this new version and you'll see it produce the squares of numbers just as the first version did.

Although the (while) version works, it is more inefficient than the example in Chapter 6 using multiple rules. CLIPS is really designed for efficiency with multiple rules. Trying to make a procedural program out of CLIPS negates the advantages of a rule-based language. Carried to an extreme, you could write a one-rule program in CLIPS using (while) and (if then else). However, this completely defeats the purpose of the language for efficiency with multiple rules.

As another example of the (while) syntax, consider a program to print the mean or average of numbers. The following program shows how it can be done using a single rule. Enter and run the program to compute the average of the five numbers, 1, 2, 3, 4, and 5.

```
(defrule input-max
   (start-fact)
=>
   (printout "How many numbers? "crlf)
   (bind ?max (read))
   (bind ?count 1)
   (bind ?sum 0)
   (while (<= ?count ?max)
          (printout "Number " ?count " = ? " crlf)
          (bind ?number (read))
          (bind ?sum (+ ?sum ?number))
          (bind ?count (+ ?count 1)))
   (bind ?answer (/ ?sum ?max))
   (printout crlf crlf "Mean = " ?answer crlf))
```

Notice that only one rule was necessary and no facts were asserted.

Not Meanwhile

Although the previous rule for the mean works, it would be convenient if you didn't have to count in advance how many numbers there are. That is, it would be nice if the program kept count of the numbers that were read until a **sentinel** was found. A sentinel is a special symbol that indicates the end of input.

The following version of the program uses the sentinel "done" to signify end of input. Enter and run for numbers 1, 2, 3, 4, and 5. Then type "done" when you're asked for the sixth number (do not enter the quotation marks). The computer will then print the mean.

```
(defrule not-mean-while
   (start-fact)
=>
   (bind ?sum 0)
   (bind ?count 1)
   (printout "Number 1 = ? " crlf)
   (bind ?number (read))
   (while (! (eq ?number done))
      (bind ?count (+ ?count 1))
      (bind ?sum (+ ?sum ?number))
      (printout "Number " ?count " = ? " crlf)
      (bind ?number (read)))
   (bind ?mean (/ ?sum (- ?count 1)))
   (printout crlf crlf "Mean = " ?mean crlf))
```

The while-loop reads input and assigns it to ?number until a "done" is entered. Notice the "not" function, "!", in the while-loop test (! (eq ?number done)). This test says to execute the actions in the body of the while-loop while ?number is not equal to "done".

The "eq" function was used instead of the "not equal" operator "!=" because "!=" can only be used with numbers, while "eq" can be used with both numbers and strings. In fact, you must use "eq" for strings since CLIPS has no other function to check the equality of strings.

An alternative approach to a string sentinel would be to use a special number such as 1e38 as sentinel so that you can use "!=". This sentinel is picked because it's a number that's unlikely to occur in data. Of course, if a number like 1e38 is likely to occur, you should pick another.

As another simple example of (while), the following rule is very useful if you have a lot of (assert) statements to make and don't want to use a (deffacts).

```
(defrule auto-assertions
   (initial-fact)
=>
   (printout "Assertion ? " crlf)
   (bind ?response (read))
   (printout "Asserted " ?response crlf)
   (while (! (eq ?response done))
          (assert (?response))
          (printout "Assertion ? " crlf)
          (bind ?response (read))
          (printout "Asserted " ?response crlf)))
```

When you run this rule, it will assert facts until you type "done". You'll find it convenient to save utility rules like this and incorporate them in programs just as you'd use a subroutine library in other languages.

As another example of the while statement, consider the program to calculate which numbers are perfect squares that was discussed in Chapter 8. Instead of requiring a separate rule, make-numbers, to assert the numbers, the program can be written as two rules.

```
(defrule make-numbers
   (initial-fact)
=>
   (printout "Initial number ? " crlf)
   (bind ?initial (read))
   (printout "Final number ? " crlf crlf)
   (bind ?final (read))
   (while (<= ?initial ?final)
      (assert (number ?initial))
      (printout ?initial crlf)
      (bind ?initial (+ ?initial 1))))

(defrule perfect-squares
   (number ?x)
   (number ?y)
   (number ?z&:(= (* ?z ?z) (+ (* ?x ?x) (* ?y ?y))))
=>
   (printout "x = " ?x " y = " ?y " z = " ?z crlf))
```

A Rule Of Sorts

Suppose you wanted to write a single rule that would sort a list of numbers, where some of the numbers might be duplicates. Can it be done?

The answer is yes as the following program shows. The first rule, called input-numbers, allows the user to input the numbers to be sorted and asserts them as facts. The second rule, called sort, does all the sorting. Actually, the sort rule just prints out the numbers in sorted order. However, it would be easy enough to assert the sorted numbers as they are printed out and so produce a sorted list of facts.

Enter the program and run for the numbers 3, -2, 1, 4, 1, and 0. Then type "done" (don't type in the double quotes) to stop input and the sort rule will print out the sorted numbers.

```
(defrule input-numbers
   (initial-fact)
=>
   (bind ?count 1)
   (printout "Number 1 = ? " crlf)
   (bind ?number (read))
   (while (! (eq ?number done))
      (assert (?count ?number))
      (bind ?count (+ ?count 1))
      (printout "Number " ?count " = ? " crlf)
      (bind ?number (read))))

(defrule sort
   ?biggest <- (?count1 ?number1)
   (not (~?count1 ?number2&:(> ?number2 ?number1)))
=>
   (printout ?number1 crlf)
   (retract ?biggest))
```

The input-numbers rule puts a different identifying number based on ?count in front of each number to be asserted. That way duplicate numbers can be entered. For the sample numbers, the asserted facts would be

```
(1  3)
(2   -2)
(3   1)
(4   4)
(5   1)
(6   0)
```

The "number" field is not neccessary in this example because the program uses facts with two fields for storage of the count and number. So (initial-fact) can never match because it has only one field.

Notice how count gives each of the numbers a unique identifier. Otherwise, the two number 1's could not be asserted. Of course, if you don't have any duplicate numbers, the ?count is not necessary.

In the sort rule, the first conditional element matches to any number. However, the second conditional element is more picky. It says that there is no number greater than the number matched by the first conditional element. If you think about it, this is just the definition of the largest number in a list: there is no greater number.

Let's look at the second conditional element in more detail to understand how it works. The second conditional element requires several things. First, the count identifier must be unequal to ?count1. This prevents a fact from matching itself. Second, ?number2 must be greater than the first number, ?number1, that was matched.

Suppose both of these requirements are met in the second conditional element. Then the logical not in front inverts the result and says it's not a match after all. So if you think you've got it tough, imagine what it's like to be an inference engine and told that a success is really a failure.

The sort rule finds the biggest number in the fact-list and prints it out. Then the (retract) removes the biggest number and the sort rule fires again to find the new biggest number. The new biggest is printed out, then retracted and the sort rule fires again. This process continues until there are no more numbers left in the fact-list.

If You Don't Care

In the Sort Program, the count numbers were only used as internal identifiers of facts. That is, the value of the numbers was not important. Only the fact that each count number was unique was important in using the counts as identifiers.

If you don't care what the actual value is, then CLIPS has just the thing for you. The **gensym function** returns a unique symbolic atom every time it's called of the general form

genX

where X represents a different number each time gensym is called. The (gensym) numbers always start at 1 when CLIPS is first started and increment by 1 each time (gensym) is called until you exit CLIPS.

As an example of (gensym), enter and run the following program to make ten calls of (gensym).

```
(defrule make-gensym
   (initial-fact)
=>
   (bind ?count 10)
   (while (> ?count 0)
      (bind ?sym (gensym))
      (printout ?sym crlf)
      (bind ?count (- ?count 1))))
```

You'll see the following output.

```
gen1
gen2
gen3
gen4
gen5
gen6
gen7
gen8
gen9
gen10
```

Now run the program again and the symbols will start with gen11.

You can also start (gensym) off with a value that you specify with the **setgen** function. The (setgen) must be used in a **call** to an **external function** on the RHS. For example, modify the make-gensym rule as shown below and run. Now the (gensym) will start off at 100.

```
(defrule make-gensym
   (initial-fact)
=>
   (bind ?count 10)
   (call (setgen 100))  ; add this line
   (while (> ?count 0)
      (bind ?sym (gensym))
      (printout ?sym crlf)
      (bind ?count (- ?count 1))))
```

Now let's apply (gensym) to the sorting program. Instead of using ?count and having to do arithmetic to obtain new identifiers, let (gensym) do the work. The following program is the (gensym) version. Enter and run it for some numbers and you'll see the program sorts just as good as before.

```
(defrule input-numbers
   (initial-fact)
=>
   (printout "Number ? " crlf)
   (bind ?number (read))
   (while (! (eq ?number done))
      (assert (=(gensym) ?number))
      (printout "Number ? " crlf)
      (bind ?number (read))))

(defrule sort
   ?biggest <- (?count1 ?number1)
   (not (~?count1 ?number2&:(> ?number2 ?number1)))
=>
   (printout ?number1 crlf)
   (retract ?biggest))
```

The only difference between the two sorting programs is that the (gensym) version does not print out identifying numbers as you input the numbers.

Whiles In Whiles

As long as we're enhancing the program, let's go all the way and have it ask if we want to do another. Although this could be done by having another rule triggered when the not-mean-while rule ends, let's be elegant and do everything within one rule.

The following program shows **nested while-loops.** One loop, called the **inner loop,** is entirely contained within another, the **outer loop.** Run for an input of 1, 2, 3, 4, and 5. Type "done" and you'll see the result mean. Then type "n" and the program will ask for a new set of numbers for a mean calculation.

```
(defrule stop-not-mean-while
   (start-fact)
=>
(bind ?response n)
(while (eq ?response n)
   (bind ?sum 0)
   (bind ?count 1)
   (printout "Number 1 = ? " crlf)
   (bind ?number (read))
   (while (! (eq ?number done))
      (bind ?count (+ ?count 1))
      (bind ?sum (+ ?sum ?number))
      (printout "Number " ?count " = ? " crlf)
      (bind ?number (read)))
   (bind ?mean (/ ?sum (- ?count 1)))
   (printout crlf crlf "Mean = " ?mean crlf)
   (printout "Stop program ? (y or n) " crlf)
   (bind ?response (read))))
```

The inner loop will keep asking for input data until the user inputs "done". Then the program will print out the mean and ask if another is to be done. If the user answers "n", the program ends since the outer while-loop test, (eq ?response n), is true.

Although the if and while are very powerful, they should be used sparingly. If you're using them a lot, your program is becoming sequential and an expert system language may not really be the best choice for your application.

The Missing Evidence

Sometimes the absence of an item reveals more information than it's presence, especially in murder mysteries. CLIPS allows you to specify the absence of a fact as a conditional element on the LHS using the **logical not function.**

The syntax of the conditional element with (not) is

```
(not (pattern))
```

where "pattern" represents the fact that is to be matched.

As a simple example, think back to the problem of telling a robot when to cross the street. The rules which tell when to walk are the following.

```
If there is a walk-sign that says walk
then walk
```

```
     If there is no walk-sign and the light is green
     then walk
```

The (not) function can be conveniently applied to the simple rules above as follows.

```
(defrule walk-sign-walk
   (walk-sign walk)
=>
   (printout "Walk" crlf))

(defrule no-walk-sign
   (light green)
   (not (walk-sign ?))
=>
   (printout "Walk" crlf))
```

Notice that a wildcard question mark is used for matching the second atom of (walk-sign) in the no-walk-sign rule. The rule won't be triggered on just (walk-sign) since we assume the second atom contains the status of the sign. That is, if there is a (walk-sign), it will be either (walk-sign walk) or (walk-sign dont walk).

Enter these rules and assert (light green). Check the agenda and you'll see that only no-walk-sign will be triggered. Now assert (walk-sign walk) and you'll see that no-walk-sign will be removed from the agenda and walk-sign-walk will be put on the agenda.

It's important to realize that the no-walk-sign rule is not the same as a rule with one conditional element like

```
defrule green-light
   (light green)
=>
   (printout "walk" crlf))
```

The green-light rule will be triggered whether there is or is not a (walk-sign) fact. However, the no-walk-sign rule will be triggered only if there is not a (walk-sign). The difference between the rules is subtle but important.

A (not) can only be used to negate one conditional element at a time. You must use a separate (not) for each conditional element to be negated.

Problems

1. Rewrite problems 1 - 3 of Chapter 6 using a (while) statement to do all the calculations in one rule.

2. Write a program with one rule using nested (while) loops to generate the multiplication tables from 1 to 9. The output of the program should be in the form

```
1 x 1 = 1
1 x 2 = 2
1 x 3 = 3
```

and so forth.

3. Write a program to control the operation of a portable electric generator driven by a Diesel engine. A simplified diagram is shown in Fig. 10-1. The generator produces electricity if the engine is working properly. The engine is working properly if there is a normal amount of oil for lubrication, fuel, and water for cooling.

Some of the possible abnormal conditions and corrective actions are shown in the following table. Based on the diagram, add other abnormal conditions and corrective actions to the table. Each action should also print out a message of the actions taken and the conditions that triggered the action.

Conditions		Actions
Oil pressure is low > 50 and <= 100	:	Check Reserve Tank ORT-1
Oil pressure too low <= 50	:	Switch on Backup Oil Pump OP-2 Check Oil Pump OP-1
Water temperature high > 125 and <= 165	:	Check Water Pump WP-1
Water temperature too high > 165	:	Turn on WP-2

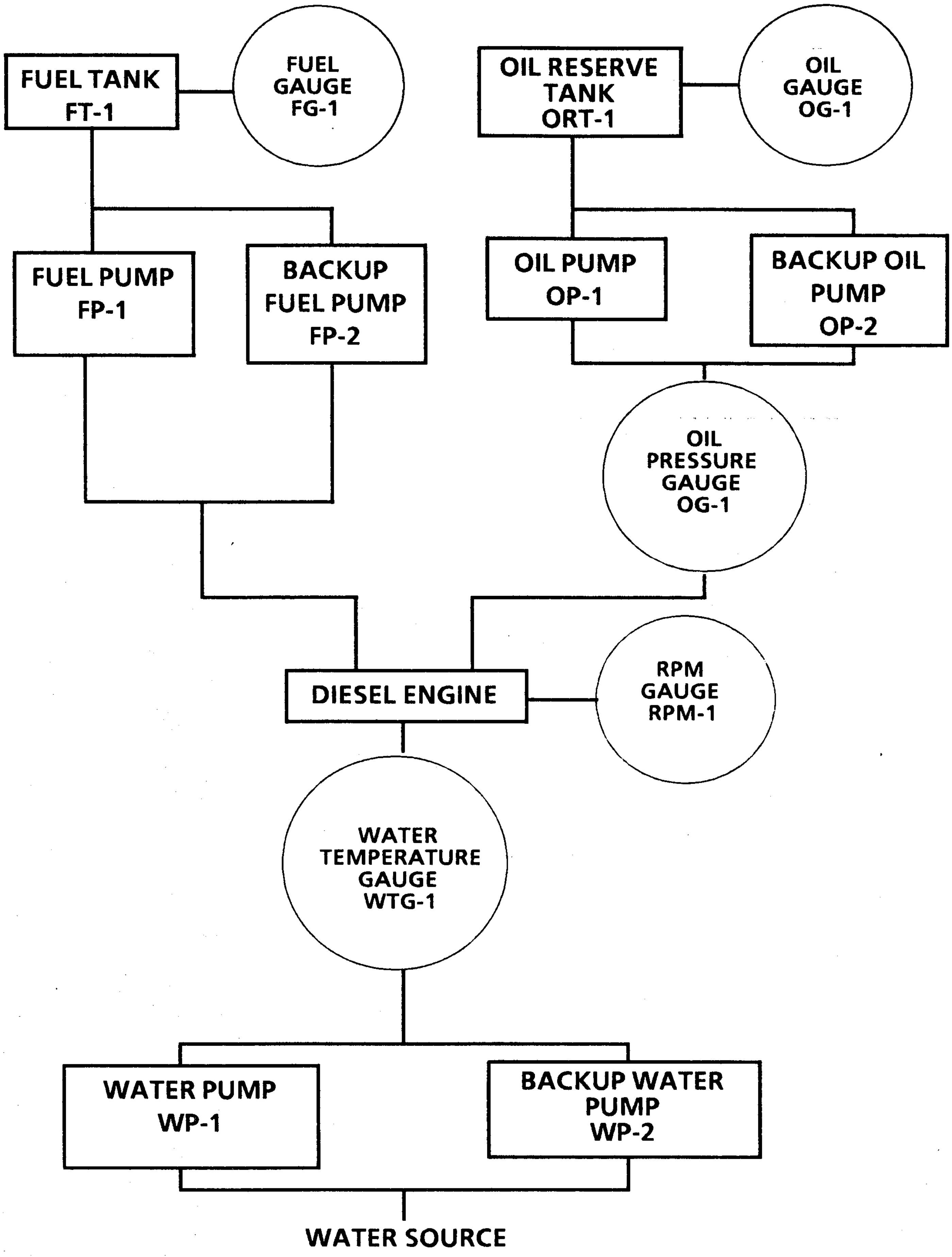


FIGURE 10-1 DIESEL ELECTRIC GENERATOR

```
Electric demand is high    : Increase RPM to 2000
> 900 and <= 1000

Electric demand very high : Increase RPM to 2500
> 1000 and < 1100
```

Write a program that asks the user to input abnormal conditions. That is, the user can input that the oil pressure is low or that electric demand is very high. The program will then print out the corrective action and then ask the user for new conditions. Also, include rules for excessive conditions such as no fuel that will cause a shutdown.

In order to simplify input of abnormal conditions by the user, display all of the possible conditions in the form of a menu. For example, part of the menu might be

```
Oil pressure low-------1   Water temperature high-------3
Oil pressure too low---2   Water temperature too high---4
Exit program----------99
```

The user would input the menu numbers of conditions.

Assume that a reading never just jumps to a too high or too low condition. That is, the condition will first pass through a low or high before becoming too low or too high. Thus all of the appropriate actions for the low or high will be taken before the too low or too high conditions.

Some useful information is the following.

```
Fuel low     : 125
Fuel too low : 50
```

The engine will not operate if it is out of fuel, oil, or water.

If any one backup system is operating, the system cannot meet an electric demand that is very high and so a brownout will occur. Print out a brownout warning message.

If more than one backup system is operating, and electric demand is very high, a blackout will occur. Print out a blackout warning message.

If the electric demand is greater than or equal to 1100, a brownout will occur because the RPM cannot be increased beyond 2500. Print out a brownout warning message.

Chapter 11 Filing Around

Up to this point, you've seen how CLIPS can work with facts that have been entered from the keyboard or rules. However, in some cases you may wish to read or write facts to disk. In this chapter, you will learn how to use CLIPS for file I/O of facts.

Opening Things Up

Before a file can be accessed for reading or writing, it must be opened using the **open** function. The number of files that can be opened at once is dependent on your operating system and hardware.

The general form of the open function is

(call (open "<file-name>" <file-ID> ["<file-access>"]))

As an example,

(call (open "animals.dat" animals "r"))

The open function is called with a call function because you really want the side-effect of opening a file rather than the value returned by the open function.

The first argument of open, "<file-name>", is the name of the file on your computer. Generally, this will be the physical file name unless you've re-assigned the name using an operating system command. For the example shown, the file-name is animals.dat.

The second argument, <file-ID>, is the **logical name** under which CLIPS knows the file. Although the logical could be the same as the file-name, you may want to rename it. For example, if the file-name is animals.dat, you may want to rename it without the period so that you don't have to keep typing double quotes around the name everywhere you use it in CLIPS. So you might rename animals.dat as animals or by a different name such as data, my-data, input, or any other valid name.

Another advantage of a logical name is that you can easily substitute a different file-name without making major changes in the program. For example, you may want to read data from animals.dat for awhile and then read data from a file called birds.txt. Only the open statement need be changed to accomplish this.

The third argument, "<file-access>", is optional as indicated by the square brackets around it. The four possible modes of file-access are

Mode	Action
r	**read access only**
w	**write access only**
r+	**read and write access**
a	**append access only**

The default value of file-access is "r".

As an example of the open function,

```
(call (open "animals.dat" data "r"))
```

will open a file called "animals.dat" and assign it the logical name data. Since the file-mode is "r", an alternative statement is

```
(call (open "animals.dat" data))
```

Either version of the open statement can be used.

...And Closing Them Down

Once you no longer need to access a file, you should close it down. Unless a file is closed, there is no guarantee that the information written to it will be saved.

Although you can open a file when the first rule fires and close it when the last rule fires, you're taking an unneccesary risk if you don't need to access the data all the time. There may be a power outage or hardware failure that prevents the file from closing and so the data you've written to it may be lost. The disadvantage of opening and closing files in a rule is that execution is slower.

The general form of the **close** function is

```
(call (close [<file-ID>]))
```

where the optional argument specifies the file to be closed. So

```
(call (close data))
```

will close the file known to CLIPS by the logical file-name animals. The statements

```
(call (close input))
(call (close output))
```

will close the logical files named input and output. Note that separate statements are necessary to close specific files.
   The default case in which no file-ID is specified

```
(call (close))
```

will close all open files.
   Let's try an example now of opening and closing files. First, we need to create the files to be accessed. Create one file called "animals.dat" with the data shown below.

```
duck goose   ganders
geese
ducks
```

Now make a file called "vegtable.dat". You don't have to put any data in this file. Notice that the filename is limited to eight characters because of limitations on MSDOS filenames. Of course, you can also use any filenames that are valid on the system you are using. Just modify the names in the following program to open and close the appropriate files.

```
(defrule read-file
   (initial-fact)
=>
   (printout "Call 1" crlf)
   (call (open "animals.dat" data1 ))
   (printout "Call 2" crlf)
   (call (open "vegtable.dat" data2))
   (printout "Call close" crlf)
   (call (close))
   (printout "Files closed" crlf))
```

When you run this program, CLIPS will print the appropriate messages as it opens and closes files.

A Graceful Exit

Although the previous program did nothing but open and close files, it was important to remind you of the necessity to have both an open and close statement. Before you write practical programs that read and write to files, you should be very aware of these requirements. In particular, if you do not issue a command to close a file, the data you have written to it may be lost.

CLIPS will not prompt you to close an open file. The only safeguard built into CLIPS for closing files that have been inadvertantly left open is when you issue an (exit) command. CLIPS will close all open files when an (exit) is issued.

However, a problem arises if you terminate CLIPS with an interrupt control character such as Control C or the appropriate interrupt for your computer. Depending on your operating system, the data you have written to open files may not be saved. That's why it's always a good idea to exit CLIPS with an (exit) rather than an interrupt. A graceful exit will save any data in files inadvertantly left open.

Well, Read My File

Let's take a look now at reading data from a file. The function to read data from a file is the familiar (read). The only new thing that you have to do is specify the opened file to read from as the argument of (read).

The following program shows how you can read data from a file.

```
(defrule read-file
   (initial-fact)
=>
   (printout "Name of file to read ? " crlf)
   (bind ?name (read))
   (call (open ?name data))
   (bind ?input (read data))
   (printout "Data read from file " ?name " is " ?input crlf)
   (assert (data ?input))
   (call (close))
   (printout "File " ?name " is closed" crlf))
```

This program shows a general way to read from a file. Instead of hard-coding the file name into the program, you can bind the name of the file to be opened. Also, notice that the default form of open is used because the file is opened for reading only.

Run the program to read data from the "animals.dat" file. Be sure to input the double quotes around "animals.dat" since CLIPS will not accept the period as part of a valid name.

The output from the program will be the first atom stored in the file, duck. Just as in inputting data from a keyboard, CLIPS will read one atom at a time. Since a space or carriage return acts as the terminator of an atom, the (read data) will stop reading after the first atom, duck, is read.

Give Me More

In order to read more than one atom, you need to set up a loop. This loop can be written by having one rule trigger another. Another way is to use a while-loop.

However, before multiple values can be read in by a loop, a very important question needs to be addressed. The important question is how do you stop the loop? The loop should not try to read past the end of file or the operating system will issue an error message. In order to prevent this, CLIPS returns a **EOF** symbolic atom if you try to read past the end of file.

The following program illustrates one way of reading in multiple values from a file.

```
(defrule read-file
   (initial-fact)
=>
   (printout "Name of file to read ? " crlf)
   (bind ?name (read))
   (call (open ?name data))
   (bind ?input (read data))
   (while (!(eq ?input EOF))
      (printout ?input crlf)
      (bind ?input (read data)))
   (call (close))
   (printout "*** File " ?name " is closed ***" crlf))
```

Enter and run this program on the file "animals.dat". You'll see that it does read and printout all the data in the file.

Printing To Files

Besides reading from files, CLIPS allows printing to files with the **fprintout** function. The first argument of (fprintout) is the logical filename. Following this is the data to be printed to the file. The following program shows an example of printing to a file.

```
(defrule print-to-file
   (initial-fact)
=>
   (printout "Name of file to print to ? " crlf)
   (bind ?name (read))
   (call (open ?name data "w"))
   (printout crlf "*** Enter data to file " ?name
             crlf "    Use EOF to stop input ***" crlf)
   (bind ?input (read))
   (while (!(eq ?input EOF))
      (fprintout data ?input crlf)
      (bind ?input (read)))
   (call (close))
   (printout "*** File " ?name " is closed ***" crlf))
```

Unlike (read), which reads only one atom at a time, you can use (fprintout) with as many atoms as you want. For example,

```
(fprintout data "This is being printed to a file" ?input crlf)
```

will print all the atoms after data to the file.

A Standard Change

If the first argument of (fprintout) is the special logical filename "t", then output will be sent to the standard output device. For example, in the previous program, change the (fprintout) action in the while-loop to

```
(fprintout t ?input crlf)
```

and run again.
   Instead of printing to a file, the output will go to the standard output device. Usually, the standard output device will be the terminal screen and so you'll see the output of the (fprintout) on your screen. However, depending on your operating system, you may be able to issue an operating system command to change the standard output to another device such as a printer or modem port. In that case, the output of (fprintout) will go to your printer or modem.
   The (read) function also has a "t" option so that you can read from the standard input device. Normally, the standard input device is the keyboard, but your operating system may allow the standard input to be redefined just like the standard output. For example, you may be able to redefine the standard input as a modem port or lightpen port.

Chapter 12 The Great Expansion

In the preceeding chapters of this book, you've learned the syntax of CLIPS. However, CLIPS has an additional feature that greatly extends its capabilities by allowing you to define your own functions. If you are an experienced C programmer, you can learn from this chapter how to make your own customized version of CLIPS.

Random Problems

In order to make your own user defined functions, you should have knowledge of C, since CLIPS is written in C. The following description of how to add a function assumes you know the C language and how to compile programs in C. The description is for an implementation on an IBM personal computer using the Lattice C compiler, and also for a UNIX based system. However, it should be understandable for other C implementations also.

If you are using an IBM personal computer, you will probably not be able to compile CLIPS on a floppy disk system because the object code files are so large. Even using RAM disks and a 640 K machine will still give trouble in compiling. A hard disk is really necessary.

The user defined function to be added to CLIPS is a random number generator. There are many cases in which a random number generator is of use. Since a random number generator is not provided in the CLIPS syntax or in the math library, let's write a user defined function and add it to CLIPS.

As a first step in adding the random function, create a file called rand.c or a name appropriate to your system. The random number generator used will be the simple random number function provided by the Lattice C compiler, **rand**.

```
#include <math.h>
#include <stdio.h>
#include "clips.h"

float my_rand()
   {
       extern int rand();
       return ((float) rand());
    }
```

The name "my_rand()" can be any name you want. The rand function must be declared extern because it is not defined in the rand.c file. Normally, the rand function returns integers. But since CLIPS only uses floating point numbers, it is cast as a float in the return statement.

The second step in adding the random function is to modify the main.c file. The statements which have been modified are shown with comments that include an "!" as in "/***!" in the USRFUNCS() section at the end.

```
/*  CLIPS: main.c (V3.01) Last modified 7/21/86    */

#include <stdio.h>

/*******************************************************************/
/* MAIN: Start execution of CLIPS.  This function must be          */
/*   redefined in order to embed CLIPS within another program.     */
/*   Example of redefined main:                                    */
/*     main()                                                      */
/*       {                                                         */
/*        init_clips();                                            */
/*            .                                                    */
/*            .                                                    */
/*            .                                                    */
/*        process_data();                                          */
/*        run(-1);                                                 */
/*        evaluate_data();                                         */
/*            .                                                    */
/*            .                                                    */
/*            .                                                    */
/*        final_results();                                         */
/*       }                                                         */
/*******************************************************************/
main()
  {
   printf("        CLIPS (V3.01 7/21/86)\n");
   init_clips();
   command_loop();
  }
```

```
/*****************************************************************/
/* USRFUNCS:  The function which informs CLIPS of any user    */
/*   defined functions.  In the default case, there are no    */
/*   user defined functions.  To define functions, either     */
/*   this function must be replaced by a function with the    */
/*   same name within this file, or this function can be      */
/*   deleted from this file and included in another file.     */
/*   User defined functions may be included in this file or   */
/*   other files.                                              */
/*   Example of redefined usrfctns:                            */
/*     usrfuncs()                                              */
/*       {                                                     */
/*        define_function("fun1",'i',fun1);                    */
/*        define_function("other",'f',other);                  */
/*       }                                                     */
/*****************************************************************/

 /***! THESE STATEMENTS ARE ADDED FOR THE RANDOM FUNCTION !***/

usrfuncs()
  {
     extern mathfctns();  /***! includes the math library !***/
     extern float my_rand();    /***! add random function !***/

     mathfctns();           /***! includes the math library !***/

     define_function("rand",'f',my_rand); /***! add random !***/
  }
```

The above changes also add the math library functions called mathfctns(). Strictly speaking, it's not necessary to add the math library in this example for the random function. However, it's most likely you'll want to use both the math library and the random function generator and so both are shown.

Lotta Links

Once you have these two files created, it's time to compile them. On a UNIX system, enter

```
cc -c rand.c
```

To compile the rand.c file on a Lattice C compiler, you will have to enter a command specific to the directory storage of your files. The general form of the command is

```
lcl =4096 rand.c -ml -n
                 -i { Lattice include files }
                 -i { CLIPS include files }
```

where
4096 increases the stack size for the compiler
-ml is for the large memory model option
-n allows long variable names, greater than nine characters

Depending on your directory structure, the Lattice include files "i option" may be

```
-ic:\lattice\includes\
```

if they are on drive c: in the directory \lattice\includes\.

Likewise, if the CLIPS include files are on drive c: in the directory \clips\source\, then the "i option" command is

```
-ic:\clips\source\
```

Compile the main.c file in the same manner.

The output of the compiler is an object code file which is then linked to the other CLIPS object files. Note that it is not neccessary to re-compile the other object files such as parser.o, clips.o, and so forth since they have not been changed.

In a UNIX system, you should first copy all object versions of CLIPS files to your directory. This is a precaustion in case your linker cannot follow the path names to object files in other directories. The UNIX command to link the object modules is

```
ld -o myclips rand,clips,main,math,sysdep,npsr,parser,usrint
```

where myclips is the name of your customized version of CLIPS with the random function.

The command for the linking with the Lattice C compiler will depend on the directories your object files are stored in. The general form of the link command is

link { Lattice large memory link originator }

The files to be linked are

**clips.o,rand.o,main.o,math.o,sysdep.o,npsr.o,
parser.o,usrint.o,myclips,null,**

where myclips is the name of your customized version of CLIPS, and null is an option to the linker which says not to create a memory map. Note that the comma after the null is required. If you do not specify a name for the executable code, for example myclips, the linker will use the first argument, clips, as the name.

In addition, you will also need to link the

{ Lattice large memory math library }
+
{ Lattice large memory function library }

The output should be the new CLIPS with the user defined random function.

A Good Execution

To test out the random function, start up CLIPS and enter this test program.

```
(defrule random-test
   (initial-fact)
=>
   (bind ?count 0)
   (while (< ?count 10)
      (bind ?rnd (rand))  ; call the random number function, rand
      (printout ?rnd crlf)
      (bind ?count (+ ?count 1))))
```

When you run this program, it will generate 10 random numbers.

Notice that a separate file was created for the random number function. Although you could add the appropriate statements to the math library file, math.c, it is definitely not a good idea. As new releases of CLIPS are given out, the math.c file will be updated and so you would have to modify it again. There is also the danger of your having a non-standard version of the math library. It is preferable to keep your functions in a separate personal math library file.

The math.c file has some routines for testing the number of arguments to a function that you may find useful if you add functions with arguments to your personal math library. Note that the random function does not use arguments and so no testing is really neccessary. However, you may want to include argument testing in other functions.

Index

DISTRIBUTION FOR JSC IN 86-FM-25

NASA - Hdqts.
DE/L. Clark
RC/M. Montemerio
R. Grumm
L. Holcomb
C. Roberts

NASA - JSC
DF74/W. Bates
EE8/W. Marker
EF/K. Healey
EH23/R. Farley
EH5/D. Platt
B. Hendrix
FA/R. Berry
J. Garman
FR12/J. Raney
G. Weisskopf
FR4/J. Fisher
J. Snyder
FM/E. Lineberry
FM2/D. Heath
FM7/R. Brown
FM72/R. Savely (10)
S. Richardson
M. Boarnet
L. Wang
G. Riley
C. Culbert (40)
J. Villarreal
P. Mitchell
E. Henry
R. Lea
G. Martinez
E. Grice
J. Giarratano (20)
FS92/T. Marohn
JM2/Technical Library (3)
JM86/Distribution Operations (15 + extras)
JM88/Documentation (2)
KC2/H. Kaupp
PB/M. Holley
SR/J. Erickson
J. Malin

NASA - ARC
244-7/~~H. Lum~~
P. Friedland
~~P. Cheesman~~
W. Gevarter
T. Grant
S. Starks
W. Erickson
M. Schwartz
V. Frederick
P. Nachtsheim
239.3/E. Palmer

NASA - GSFC
514/J. Liebowitz
520/J. Dalton
522.1/W. Truszkowski
T. Ames
L. Hull
D. Perkins
531.1/J. Rash

NASA - JPL
180-701/D. Pivirotto
238-641/ Tausworthe
201-214/E. Biefeld
S. Grenander
241-210/K. Blom
506-328/K. Volkmer
510-202/D. Atkinson
M. James

NASA - KSC
SE-FSD/J. Ragusa
J. Jamieson
A. Beller
SC-LPS-13A/J. Rogers
PT-FPO/T. Davis

NASA - LaRC
113/J. Creedon
152D/A. Meintel
N. Sliwa
156A/K. Abbott
P. Schutte
246/J. Rogers
288/K. Willshire

NASA - LeRC
301-3/K. Faymon
49-7/L. Berke
5-7/B. Steinburg

NASA - MSFC
EB44/K. Fernandez
C. Wang
EB12/D. Weeks
EB24/E. C. Smith
M. Freeman

NASA - WSTF
RF111/T. Estes

OMNIPLAN - Houston
Editing
17226 Mercury Drive
Houston, TX 77058

Capt. J. P. Squatrito
2nd Space Wing/DO3
Falcon Air Force Station, CO 80912

Dr. N. S. Rajaram
University of Houston/University Park
Mechanical Engineering Department
Houston, TX 77004

Dr. C. Dede
University of Houston/Clear Lake City
2700 Bay Area Blvd.
Houston, TX 77058

Dr. R. Allen
University of Houston/University Park
Mechanical Engineering Department
Houston, TX 77004

Any additions, deletions, or corrections to this distribution should be directed to S. Richardson, FTS 525-3921 or telemail id SRICHARDSON at NASA-JSC, mail code FM72.